Highway construction in Wireless Sensor Networks

Antoni Segura Puimedon

Faculty of Informatics of Barcelona

BarcelonaTech

A thesis submitted for the degree of

Computer Engineering

2011 May

Abstract

Wireless sensor networks are a rapidly growing field of study with many open research topics. The aim of this project is to build a hierarchy of clusters in wireless sensor networks and to communicate them through distinguished paths. Those paths are known as highways, and simplify higher level node inter-communication while reducing energy and memory requirements. To achieve this goal several distributed algorithms were designed and tested either in simulators or in real hardware. The message delivery rate, through highways, measured in hardware was close to 70% and it effectively served as base for a higher level network module to make end to end communication between every node of the connected network. This opens a way for the development of more algorithms to make wireless sensor networks communications on large deployments effective and trouble less.

To my sister, that listened to the whole algorithm reviewing process and, despite my obfuscation, understood it and helped me with it and to Lenka and my parents, that gave me a lot of moral support when things were not going right and lifted my spirits in such a way that gave me strength and conviction to give always my 120%, and had the patience to put up with my obsessed self. Also to the Lord, for giving me strength and determination in difficult times.

Acknowledgements

I would like to acknowledge and thank the guidance, support and confidence provided by Maria and Jordi; the wiselib 101 I got from Victor as well as all his help in coding the first version of this algorithm in a very enjoyable pair programming 5 days session. I'd also like to mention the help provided by Tobias, Konstantinos and Przemek.

Contents

List of Figures

List of Tables

LIST OF TABLES

1

Introduction

The work that describes this document is developed under the support and scope of two European projects from the 7th Framework of the European Union, namely FRONTS [1] and WISEBED [2]. In the following sections, the reader will find a description of what this two projects are about and how they relate to this thesis.

1.1 FRONTS

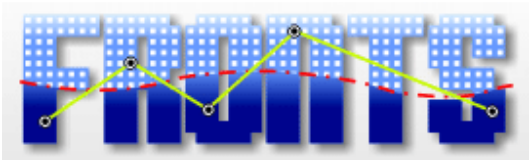


Figure 1.1: The FRONTS project logo

FRONTS is a project directed to address the reasonable expectation that, in the near future, new and revolutionary types of systems will emerge on our society. The current trends point into a direction that anticipates these emerging systems to be, in all likelihood, of massive scale, expansive, very heterogeneous, and operating seamlessly in constantly changing networked environments. These characteristics translate to an expectation, for these systems, to operate even beyond the complete understanding and control of their designers, developers, and users. Despite their perpetual adaptation to a constantly changing environment, they will be required to meet clearly-defined objectives and provide guarantees about certain aspects of their own behavior. FRONTS expects that most such systems will have the form of a large society of networked artifacts.

In order to study the possibilities and nature of these large societies of networked artifacts which are expected to evolve, if not revolutionize, the way in which we conduct our lives, the FRONTS project was commissioned. The time frame for the execution of the project was set to a three year period spanning from February 2008 to April 2011. The economic resource allocation was established to be 3.093.737€ and the coordination to be conducted by Prof. Paul G. Spirakis of *RACTI*. Eleven partners conform the FRONTS project. More details on the participating sites and their coordinators can be found on the table 1.1.

The artifacts composing these societies will be individually unimpressive: small, with limited sensing, signal processing, and communication capabilities, and usually constrained by limited energy reserves. However, by making them cooperate and transcend their individual capabilities, to become part of these large societies, it will be possible to accomplish tasks that are far beyond the grasp of contemporary conventional centralized systems. The aforementioned systems or societies should, in order to meet such high expectations, have particular ways to achieve an appropriate level of organization and integration. This organization should be achieved seamlessly and with appropriate levels of flexibility, in order to be able to achieve their global goals and objectives. And they should do this in a sensitive and proactive way to meet the current or anticipated needs of their "users". For this reason, they definitely need to adapt to the changes in their environment and change their internal organization by communicating, cooperating, and forming goal-driven sub-organizations. Our envisioned systems have an identified purpose (which depends on the application). Adaptation should continue to serve this purpose, i.e., sudden variations of external service requests or environmental physical conditions or of motion of network nodes should not stop the system from serving its goal. Instead, the system must continue to operate within the set of desired states with maintained, gracefully degraded or even improved quality of service.

The aim of the FRONTS project, thus, is to establish the foundations of adaptive networked societies of small or tiny heterogeneous artifacts. FRONTS intends to develop a level of understanding of such societies that will enable establishing their fundamental properties and laws, as well as their inherent trade-offs. This goal will be approached by working on a usable quantitative theory of networked adaptation based on rigorous and measurable gains. It is also within the intentions of this project to apply the generated models, methods, and results to the scrutiny of large-scale simulations and experiments,

from which it is expected to obtain invaluable feedback. The foundational results and the feedback from simulations will form a unifying framework for adaptive networks of artifacts that hopefully will enable the FRONTS researchers to come up with a coherent working set of design rules for such systems. In a nutshell, this project is about working towards a science of adaptive organization of large nets of small or tiny artifacts.

FRONTS goals could be summarized into:

- Providing constructive (algorithmic) distributed adaptation techniques.
- Providing laws on the effect of adaptation on the system performance, cost of distributed coordination of adaptation, incurred overhead (in terms of communication, energy) and possible trade-offs. Generation of schemes to qualify and measure adaptation.
- Investigating the limits of adaptation (how much to adapt, how long to adapt) and cases where adaptation is impossible.
- Testing the theoretical insights in practical scenarios by means of simulations and experiments.

The scale and nature of the concerning systems naturally requires them to be pervasive. Thus, FRONTS will assume that the systems subject to study have this property. This inherent characteristic, however, is a double edged sword, as it represents both a benefit and a constraint for the study of such systems.

The scope of the project is mainly that of a generic level. The project intends the foundational/modeling effort to cover a very wide range of possible scenarios that include systems made of medium to large numbers of tiny heterogeneous and communicating artifacts. Such scenarios include, for example, monitoring of earthquake regions, forests for fire protection, fluids, robot swarm organization in unknown terrains, and nodes in traffic. However, FRONTS foresees two possible use scenarios for applying the results to current real world applications: one that uses *RFID* artifacts for monitoring systems on an industrial environment and a second one that uses wireless sensor devices for monitoring traffic. These foreseen scenarios still include a rather wide range of different technical situations where:

(a) Devices use point-to-point or broadcast communication primitives in order to interact;

(b) devices interact with base-stations or organize themselves by avoiding dependence on any centralized party;

(c) there is sparse geographical distribution (with only few nodes being within the communication range of each other node) or dense geographical distributions (with many nodes communicating and interacting with other nodes) and

(d) some of the devices could be mobile.

The aforementioned conditions can and will naturally change throughout the system evolution, making it imperative that the nodes will sense the changes to the physical environment and accordingly adapt their operation, in order to maintain the performance of the system within acceptable levels. The devices will have to adapt the communication infrastructure, the energy expense, the collective internal structures and roles in their struggle to react effectively to the dynamically changing physical environments.

FRONTS foundational approach to adaptation includes effort to devise schemes to measure the quality of adaptation and the degree of optimality of adaptation. Some already foreseen measures are how fast the network adapts to the environmental changes (response time) and how much the system spends in terms of energy and communication overhead for a quick adaptation. Less obvious but also important measures include: how much to adapt (and the limits of adaptability), how much the system pays in overhead in order to stay prepared for possible future adaptations (cost of maintaining global structures, economy of energy, cost of continuous readiness and awareness).

The ability of networked societies of small artifacts to adapt is composed of two almost orthogonal dimensions, each with its own issues and objectives:

- The ability for internal continual self-organizational of the network.
  (a) Characterize the network awareness of components and adaptability to the needs and changes in the environment within the operating conditions.
  (b) Investigate the necessary technical requirements for the network to be always able to adapt, i.e, to be ready.

(c) Examine how fast it responds, in real time, to track variations in the operation of the network.

(d) Investigate the impact on the global network performance of individual entities adaption processes (how long does it take to reach a "steady state").

- The ability to adapt to environmental changes in a dynamic way. In particular, for systems deployed to achieve particular goals, this adaptability should also address the needs, constraints, and commands of its users.

  (a) Investigate the ability to adapt in cases of alerts.

  (b) Provide rules to prioritize the environmental changes (characterization of changes as major/critical where adaptation is needed, provide some thresholds).

From the previous statements, it is clear that for a society to be deemed adaptive, it needs to be composed of individual artifacts with certain capabilities. FRONTS does not plan to consider the uncommon capability of individual artifacts to alter and adapt their own hardware specifications by means of reassembling. Instead, its focus is on their capability to perform soft adaption, which affects their position and role in their society, as well as their interaction with the other individuals and the environment. The latter kind of adaption capability is to some extent technologically feasible for individual artifacts even today, in contrast to the former; what really lacks is the knowledge on how to combine the artifacts in useful adaptive nets.

To achieve the two main research goals described above, FRONTS needs to solve several scientific and technological problems, of diverse nature. The internal self-organization requires to address at least two problems: (a) how to continually adapt the communication infrastructure and (b) how to achieve "self-stability", which allows effective recovery from transient unexpected faults. This project has the belief that the second problem is of central importance because self-stabilization is an indispensable property of the systems under examination. The adaptation to the environment and to the needs of users requires to address the following problems: (a) how to achieve distributed cooperation, (b) how the system "tribes" discover and track resources, (c) how the net reacts to imposed, uncontrolled dynamism (such as externally imposed movements of the artifacts because, e.g., they follow ocean tides or are attached to

	Computer Technology Institute (GR) Coordinator: Prof. Paul G. Spirakis
	Braunschweig University of Technology (DE) Coordinator: Prof. Sandor Fekete
	Universität Paderborn (DE) Coordinator: Prof. Friedhelm Meyer auf der Heide
	University of Athens (GR) Coordinator: Prof. Elias Koutsoupias
	Ben-Gurion University of the Negev (IL) Coordinator: Prof. Shlomi Dolev
	Università di Roma “La Sapienza” (IT) Coordinator: Prof. Alberto Marchetti-Spaccamela
	Università degli Studi di Salerno (IT) Coordinator: Prof. Giuseppe Persiano
	Wroclaw University of Technology (PL) Coordinator: Prof. Miroslaw Kutylowski
	BarcelonaTech (CAT) Coordinator: Prof. Josep Diaz
	University of Geneva (CH) Coordinator: Prof. José D.P. Rolim
	Universität zu Lübeck (DE) Coordinator: Prof. Stefan Fischer

Table 1.1: Table of FRONTS partners

humans), and (d) the extremely important objective of how trust develops or emerges in the whole net or its parts.

Both kinds of adaptive abilities require to be able to cope, on one hand, with all kinds of threats, faults, and attacks, and on the other hand, to be able to establish and

maintain trust to the humans and to the other parts of the net. Adaptive security and trust in dynamic settings are tasks FRONTS needs to address in both lines of objectives of its research.

1.2 WISEBED

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

The partners in WISEBED have developed or improved applications such as the Tarwis system, 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 [3], including dozens of algorithms which run on many platforms, integration with Shawn simulator [4]...

Figure 1.2: The WISEBED project logo

The relationship between the Highways project and the WISEBED project has several aspects. The first aspect is being part of the first largely modular project performed on the Wiselib library[1]. The second one is related to being a user of the WISEBED testbeds in Lübeck, Patras and Geneva. Another collaboration was helping to manage the Barcelona testbed and providing it the testing and topology building. Finally, developing WiseML generators from the sensor data available in Barcelona Tech.

1.3 Highway construction

In wireless sensor networks, we call Highways to the communications paths to be established on top of an overlaid hierarchy of the nodes forming the topology. Highways should provide a more convenient and efficient way of communication. Research on highways' construction algorithms has been conducted on the framework of the FRONTS project, constituting one of the building blocks that formed the networking Layer (also known as Layer 1) of the final software delivery of FRONTS, as seen in figure 1.3. The main work presented by this document is precisely the design, implementation and general development of highway construction algorithms.

The concrete building block that is implemented by the highways is the hierarchy construction. The degree of involvement with the WISEBED project, which provides the software for the real hardware testbeds, has been increasing throughout the development. The first part of this involvement is in the fact that the final non priority queued algorithm is now part of the WISEBED library algorithms, also known as Wiselib, and the second part has been determined by the usage of the tools, scripts and occasional administration of the available testbeds[2] as the highway implementation started to materialize in a testable form.

This project spanned from October of 2010, when the requirements for the hierarchy construction algorithm were received, till April of 2011, after the 4.6 deliverable of the FRONTS project was delivered, although some work continued in the Barcelona testbed, in the frame of WISEBED. For more information about the tasks that composed the project and their lifespan, the reader can take a look at the Gantt and, in general, at the planning section that starts at page 89.

[1] A lengthy description of the Wiselib can be found on section 2.1

[2] The administration part is concerning only the Barcelona testbed.

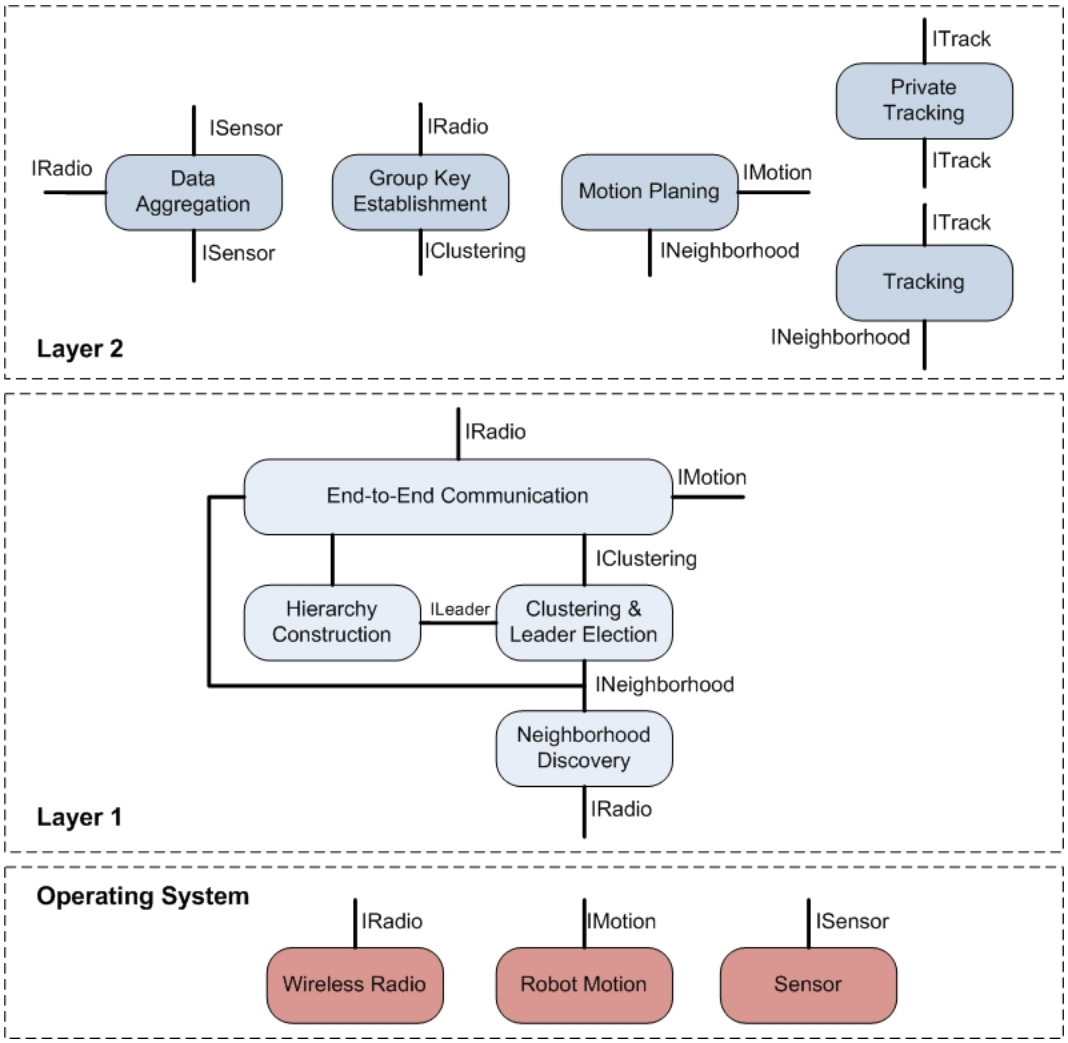


Figure 1.3: The FRONTS software architecture.

FRONTS effectively and extensively used the algorithm composition capabilities of the Wiselib in the networking Layer, in which every building block used the underlying algorithms to provide its higher level functionality to the overlying modules. In this process, the participants of the project progressively gained experience and insight on the benefits this approach can translate into. For this reason, the initial building blocks depicted in the figure 1.3 were altered to factor out a common piece of functionality that finally became an algorithm of its own for all the others to use. The factoring out proved to be effective as code size is concerned. The algorithm was a common neighborhood discovery module that replaced the previous algorithms that each site had for neighborhood management.

The networking layer structure, as seen in figure 1.3 is built on the base of the operating system (which is accessed through the Wiselib, as seen in page 20). The first module is the factored out *neighborhood discovery*, which provides a very useful interface to know the immediate neighbors of a node, and when there are changes in the neighborhood. This functionality is used by all the remaining modules, regardless of their position in the FRONTS architecture, because all of them can share the same neighborhood discovery instance, and it is much cheaper to hold a reference to it, than it is to traverse the whole stack. One could argue that in a carefully constructed design, the upper modules would simply not need this basic services, as they would be abstracted by the modules in between. However, there are a lot of use cases in which having direct access to the neighborhood saves crucial time, and in most it saves the memory that would be needed to store the abstractions.

Directly on top of the neighborhood discovery module sits the *clustering module*, which is in charge of classifying the nodes in groups and deciding a leader each for each group. The result of applying the clustering algorithm, is that, not only all the nodes become associated with a higher order entity called cluster, but they also get information of which is the node, if any, that can act as a gateway between them and the cluster leader.

Using the clustering interfaces, the *Highway module* and the *End to End communication module*. The former centers its algorithm around finding paths to connect the cluster leaders with their neighboring clusters (all the way to the leaders of those clusters), and the latter uses the clustering and the highways to take messages from any node to any node. The rationale for the End to End communication algorithm is

that if the source and target nodes belong to the same cluster, it uses a combination of clustering and neighborhood discovery to reach the destination. If the nodes are in different clusters, the node will send the message to the leader, which will send it encapsulated through a highway to the neighboring leaders, asking if they are leaders of the target node. The Highway module operation will be explained at length in the following chapters.

Over the networking layer there is the application layer, which hosts applications that make varying degrees of use of the networking layer to perform things such as secure tracking of moving nodes, group key establishment to avoid malicious interference with the normal operation of the wireless sensor network and, finally, data aggregation of collected metrics.

In terms of WISEBED involvement, it is important to note that, although the first testing for all the modules was conducted by means of simulation using Shawn, the most important experimentation was conducted on three WISEBED testbeds. The reason behind the preference for the real hardware testing and experimentation is that the FRONTS project, which defines much of the scope of the Highways project, shifted its initial goals of prospective theoretic simulations to a more current technology proof of concept approach. For this reason, the goal and experimentation presented on this document, puts the real hardware first on any decision in a very consistent basis. The testbeds used in the Highways project are in Lübeck, Geneva and Patras. For a lengthier description of the testbeds and their operation, look up page 40.

2

Technologies

In order to design and implement the highway construction algorithms that conform the present project, several technologies were used. First and foremost, one must mention the Wiselib [5] project, which is the library of algorithms and data structures developed in the scope of the WISEBED project. Using the Wiselib allowed us to target several platforms to conduct the testing and study of our algorithms. The platforms that were chosen for this purpose were the software based Shawn [6] simulator and three of the WISEBED testbeds formed by iSense [7] sensor hardware.

In the following sections, these three technologies will be introduced to the reader in varying degrees of detail according to their significance.

2.1 Wiselib

The Wiselib is library of algorithms for sensor networks that abstracts the access to the operating system resources of the targeted platforms, offering, in the majority of cases, a single interface to a resource regardless of the platform being targeted. It is obvious, that this abstracting quality can help in an inordinate amount to reduce the complexity of dealing simultaneously with simulators and real hardware, therefore setting the developer free, to concentrate almost exclusively on the problem to solve, rather than getting lost in the specifics of simulator A or sensor B. In the case of the highway construction algorithm design means being able to focus directly on the event handling.

The Wiselib library contains various algorithm classes (for instance, localization or routing) that can be compiled for several sensor network platforms such as iSense or Contiki, or the sensor network simulator Shawn. These algorithms classes are hierarchically organized in the main repository[8], thus making it straightforward to find algorithms, data structures and tools to increase the productivity of the embedded systems developer. The library is completely written in C++[9], and makes a widespread use of templates in the same fashion as Boost [10] and CGAL [11] do. The choice of the templated programming style of C++ is of central importance for the success of the Wiselib as a framework for embedded development, as it makes possible to write generic and platform independent code that is very efficiently compiled for the various platforms. However, it must be noted that the templated part of C++ has an undeniably steeper learning curve than the procedural and the classical object oriented flavors, and for this, it is recommended to learn the basics of the templates prior to tackling Wiselib development for production.

The fact that the Wiselib is implemented in C++ could misguide the developer into thinking that generic C++ libraries, containers and functions are naturally available under the Wiselib platform, although the truth is that, due to the multi platform focus of the project, the use of libraries, methods and containers is constrained by the targeted platform. This means, that if you use some of the standard language facilities like *cout*, or members of the STL, those will only be able to compile and execute as long as the targeted platform supports them. Since the capacity of sensors in a sensor network is usually very limited, the use of such constructs is typically reduced to just computer simulators of wireless sensor networks.

The library of algorithms and data structures present in the Wiselib, such as the pSTL[1] can be directly and easily integrated in your application favoring collaborative efforts or just modularity concerns, which have historically been the main use of the provided algorithms. The FRONTS project, of which this thesis project participates, is in fact the first multi site combined effort to use the Wiselib library, and it served also to test the friendliness of it towards joint projects.

[1] pSTL stands for pico STandard Library and it aims to offer some of the most popular data structures or containers of the C++ standard Library in a coding that enables them to be portable across all the targeted platforms without any change. Moreover, they tend to offer a similar interface and functionality, to decrease their friction to the typical C++ programmer

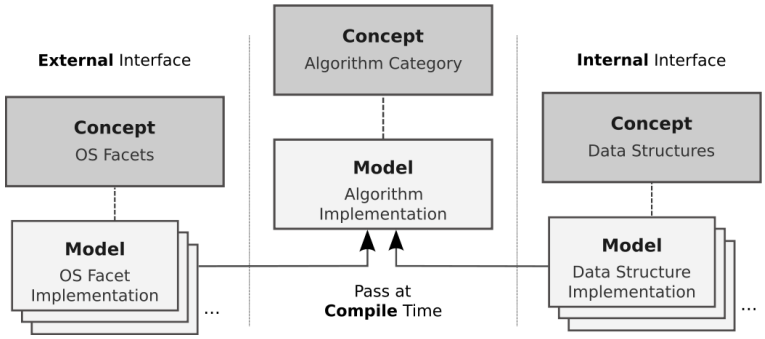


Figure 2.1: Wiselib architecture

The way in which code reutilization and modularity is regulated in Wiselib (depicted in figure 2.1) consists of a pair of abstractions called *Concepts* and *models*. The concepts define an interface and constants that, if implemented by a class, will make it a model of the concept. There are two main kinds of concepts, the ones that are related to the operating system, called *OS Facets* abd the ones that define data structures. Concepts are defined in the Wiselib documentation and thus, it is a non compiler enforced approach to interfaces. The documentation based interfaces were pitted against other kinds of patterns for interfaces and, in the Wiselib design prototyping, it was determined that the templated documented approach produces smaller (as seen in table 2.1) and more optimal machine code[1] than that achieved by more traditional approaches such as C function pointers and C++ language such as virtual inheritance.

text	data	bss	dec	hex	filename
56	4	0	60	3c	c.o
16	0	0	16	10	template.o
143	0	0	143	8f	virtual.o

Table 2.1: Table comparing different approaches to interface implementation (C function pointers, templates with concepts and virtual inheritance).

A good example of the *Concepts* way of defining interfaces is the Radio Concept, which is implemented by several models such as the *Radio*, *ReliableRadio* and *TxRadio*.

[1] A good reason for the optimality is saving the program the task of managing and containing virtual function tables.

To be compliant, these three derivative classes must implement, at least, all the defined methods, typedefs, enums, etc. as the prototype shown on the code snippet 2.1. It is clear that, just by designing and documenting a prototype, it is easy (in terms of interface adherence) to implement interchangeable modules such as different routing and clustering algorithms, which can be then used by a higher module, without changing its code.

Listing 2.1: Radio concept prototype.

```
1  concept RadioFacet
2  {
3      typedef ... OsModel;
4
5      typedef ... node_id_t;
6      typedef ... block_data_t;
7      typedef ... size_t;
8      typedef ... message_id_t;
9
10     enum SpecialNodeIds
11     {
12         BROADCAST_ADDRESS = ...,
13         NULL_NODE_ID      = ...
14     };
15     enum Restrictions
16     {
17         MAX_MESSAGE_LENGTH = ...
18     };
19
20     int enable_radio();
21     int disable_radio();
22
23     int send(node_id_t receiver, size_t len, block_data_t *
           data );
24
25     node_id_t id();
26
27     template<class T, void (T::*TMethod)(node_id_t, size_t,
```

```
        block_data_t*)>
28    int reg_recv_callback( T *obj_pnt );
29
30    int unreg_recv_callback( int idx );
31 };
```

The fact that this library provides easy-to-use interfaces to the OS which, as previously stated simplify the development process, does not completely eliminate the need to tackle occasionally the specificities of a certain platform. The main specific part one must deal with is the variability in byte length for node id variables between Shawn and the real iSense hardware, so it is handy to use the preprocessor to write platform specific code for this cases. Generally though, the OS abstraction provided by the Wiselib certainly decreases the need for handling a great deal of the low-level functionality of the targeted specific hardware platforms.

The way of building software that the proposed abstracted environment encourages is a very suitable and beneficial one when dealing with embedded software construction. It consists on coding the general application, hunting most of the bugs on the simulation environment (Shawn), were the effective cost of bug catching is order of magnitudes lower than that typical of low powered hardware like the sensors that compose wireless sensor networks, and finally testing and finishing the debugging on the real hardware. It is clear, that the fact that changes to the code are rarely needed between stages of the development cycle, except when hitting device constraints (or fine tuning that can make a difference after the code base is virtually bug free), is something not only desirable but almost indispensable when constructing part of a joint software project in an embedded project. In terms of development time, this means, that it is possible to find out a feature or a part of the algorithm to be unrealistic or unsuitable for the final product much earlier on the development cycle, thus reducing its impact over the rest of the project.

2.1.1 A glimpse into the Wiselib main resources for WSN developers

Wiselib applications normally split their functionality among two or more parts. The first being the proper application, i.e., what we want to do, and the algorithmic body used by the application. Obviously, in trivial or algorithmically simple sensor programs

one can code everything just in the application body. However, it is not generally advisable, as it detracts from modularity and the chance of the logic to be of further use for other building blocks in the way of an incorporated algorithm. Thus, the usual coding style is to program an application that includes and gives control to the main algorithm and the latter leverages its power to include other algorithms as it deems necessary or just operates by itself. Is is also usual to include several algorithms from the application which are then fed to the controlling algorithm, or even to have included algorithms as a fall-back method to perform a critical function.

From the experience of the author's and his peers on software construction with the Wiselib for wireless sensor networks, some guidelines can be extracted into which of the aforementioned code organizations are more suitable. Table 2.2 attempts to distill that empirical knowledge in an easily comprehensible way for the newcomers to Wiselib programming. The kinds to be discussed will be prototyping[1], tracer bullets[2] and modular applications.

When browsing the Wiselib code searching for resources, it is important to know how it is structured, and what principles guide the organization. The main principle is, of course, stability, due to the fact that software builders must know the degree of reliability of the software building blocks that are made available to them. Thus, in the library's trunk there are three branches, with decreasing degree of reliability. First the *wiselib.stable*, which contains the code that has been tested and known to work in at least two of the targeted platforms; then *wiselib.testing*, with the algorithms tested extensively and working in at least one of the main platforms; and finally the *wiselib.incubation* branch, which contains code that is being currently hacked together and is not guaranteed in any way to work.

Inside the two most reliable branches, the structure is almost identical and splits the code into algorithms, which contain specific code such as clustering and routing algorithms; internal interfaces, with general code constructs relevant to all the platforms

[1] Prototyping in this sense means scrapping the code away on completion, after learning the lessons the prototype was designed to answer. One good example of this would be a small broadcasting application to measure the amount of transmissions tolerable in a testbed, where tolerable stands for the threshold in which they don't decrease the ability of the other nodes to communicate. It is very useful to program a single node to jam the neighborhood and see how the algorithms react.

[2] Tracing bullets is a concept borrowed from [12, p. 48-52] which stands for building the spine of the application and have it perform in the final form some of its target features, and grow the rest of the software on it.

Wiselib app. development	Building organization
Prototyping small applications.	A single application, with all the code in a cpp file, that uses just the Wiselib standard resources or algorithms. It is self contained and cand can be useful for small tests such as figuring out the connectivity of a testbed.
Tracer bullets	Design the application by developing just one use case of the application in an algorithm which is registered by the main cpp Wiselib application. After gauging the closeness of the algorithm to the intended result, add the rest of the use cases incrementally in the same or new algorithm modules.
Modular applications	Design each of the needed algorithms separately, test them by a testing cpp Wiselib application, and finally build the composing application. This enables for a greater confidence on the building blocks that will articulate the goal achieving algorithm.

Table 2.2: Table of general Wiselib coding patterns

of the Wiselib; external interfaces, containing the resources organized and coded for each of the targeted platforms; and util, which is home to useful software constructs such as the pSTL visualization software, serialization classes, etc. The organization of the incubation branch is, on the day of writing, lacking structure and generally has a directory for every working space, but without a strong hierarchy.

2.1.1.1 Main modules and interfaces

Advancing from application and repository structure into what could be called Wiselib standard language features, we must first focus on the offered namespace. The namespace in which algorithms reside and should be placed as long as they want to be included inside the project trunk is, unsurprisingly, *wiselib*. It is in this same namespace that the main resource for programming platform independent software resides, the *OSMODEL* module. It is a Wiselib interface to the several operating systems supported by the library that picks, at compile time, the appropriate platform specific modules depending on the building target. From it the most used modules are directly derived:

- Radio
- Timer
- Clock
- Debug
- Rand

These modules will be introduced in the following paragraphs. But first, lets introduce the way in which one can get hold of them.

Listing 2.2: Getting the resources from th Operating System.

```
1 typedef wiselib::OSMODEL Os;
2 //Declaration of the Os depending pointers to modules.
3 Os::Radio *radio_;
4 Os::Timer *timer_;
5 Os::Clock *clock_;
6 Os::Debug *debug_;
7 Os::Rand *rand_;
8
```

```
9  //Obtainment of the modules by using the Facet provider
       resource.
10 radio_ = &wiselib::FacetProvider<Os, Os::Radio>::get_facet(
       value );
11 timer_ = &wiselib::FacetProvider<Os, Os::Timer>::get_facet(
       value );
12 clock_ = &wiselib::FacetProvider<Os, Os::Clock>::get_facet(
       value );
13 debug_ = &wiselib::FacetProvider<Os, Os::Debug>::get_facet(
       value );
14 rand_ = &wiselib::FacetProvider<Os, Os::Rand>::get_facet(
       value);
```

The Radio module is the first because of its central importance to wireless sensor networks, it's the one which allows us to communicate different devices among themselves and by this communication tap into a much higher functionality than a single node could ever offer. The two main uses of the radio are sending and receiving. However, each has it's own specifics which must be dealt with. Starting with the sending part, it is important to note that the transmissions can be done in a broadcast, making all the one hop neighbors listen the communication, or unicast, specifying the sole receiver of the message. Also among the important parts of the message is the length, which can be set to a maximum defined by the Wiselib (which by itself is constrained by the targeted platform).

In the code found in 2.3, the reader can find an example usage of the sending method of the extended radio resource provided by the operating system. There are three kinds of radio, the basic one, which was introduced in the code snippet 2.2, the extended radio, TxRadio, which enables the user to gather more information from the reception perspective such as the reception strength, and the reliable radio, which has mechanisms to retransmit the information sent in case an acknowledgement is not received. On to the snippet, one can notice that the type definitions are typenamed. The reason for this behavior is that this fragment of code is designed to be on an modular application, and that makes it desirable to receive the radio as a templated argument to the algorithm. Thus, to define the derived types, we must use the typename keyword. Another thing about lines 3-5 that can seem a little bit of an overhead to the novice Wiselib user is

the fact that very basic types are declared and which could be probably guessed and mapped to standard C/C++ types. I must recommend against the temptation to use such a "direct" approach as in different hardware targets, these data types are defined in non compatible ways. One example of this is the two byte Os::Radio::node_id_t found in the iSense platform and the four byte that is used by the Shawn simulator.

Listing 2.3: Sending two bytes by broadcast and unicast with the extended radio.

```
1  // Type definition of the Radio relevant types.
2  typedef typename OsModel::TxRadio Radio;
3  typedef typename Radio::size_t size_t;
4  typedef typename Radio::block_data_t block_data_t;
5  typedef typename Radio::node_id_t node_id_t;
6
7  //Declare a buffer of the maximum allowed length
8  block_data_t buffer_[Radio::MAX_MESSAGE_LENGTH];
9  //Put some data into the buffer
10 buffer_[0] = 0x1a;
11 buffer_[1] = 0x42;
12
13 //Send the message by broadcast.
14 radio_->send(Radio::BROADCAST_ADDRESS, sizeof(block_data_t)
       *2, buffer_);
15
16 //Send the message to the 0x0cda node.
17 node_id_t target = 0x0cda;
18 radio_->send(target, sizeof(block_data_t)*2, buffer_);
```

Another important part when sending information is the adjusting of the transmission power in order to save battery or to maximize the reach of the communication in cases where a node is unable to communicate with other nodes, and would, in case no facility was provided for increasing the power, be rendered unusable. To address this case, the Wiselib library offers a couple of functions that combined can dynamically alter the power, as well as a static workaround. The dynamic methods are the extended radio member *set_ power(TxPower p)* and the TxPower member *set_ dB(int db)*. The former accepts a TxPower object that can be created by the latter from an

int that specifies the desired dB of the communication in the range -30 (minimum) to 0 (maximum). The static method consists in adding a *#define DB -x* where x is a natural number from one to six which alters the communication strength. The need or the use case for a static approach arises from the need of some algorithms to tune their communication density and power to the specifics of a testbed. One good example of that is tracking applications, which depend greatly on the amount of communication among the tracking nodes.

The second part of the radio resource is the reception of messages. To receive messages, one must register a receiving method to the radio module to be called back for every message received by the node. The avid reader must have noticed that by the description just given, all the receiving methods registered to the radio module will receive all the messages to the node, potentially causing isolation and encapsulation concerns in software architectures composed by several modules with access to the radio functions. The concern is perfectly valid and needs to be properly addressed in order to work successfully with this platform. The most common solution is to reserve the first byte of the messages to enclose a message id that is checked by all the called back methods that were registered to the radio module.

In the code snippet 2.4 the reader can see both, how to register and the skeleton of a receive method. First of all, we declare the receiving method with three parameters, id of the sender, length of the message and a pointer to the message itself. Then, before paying almost any computational cost, (lines 5 and 6) we check that we are not hearing the echo of our own message, case in which we would return. After that we copy the data from the message to an internal buffer (we can't guarantee that the data pointed by *data will still be accessible inside other specialized functions, and thus, it is safer to perform a copy). Finally, we start a conditional block (lines 10-13) in which we check if the first byte of the data matches some of our defined message types (it is recommended to use *enums* instead of defines for debugging purposes, because while debugging the simulator execution, one would be offered the identifier of the value instead of some "magic" number). It is important to remember that if the amount of messages and modules is large, one should not to attach an *else* clause with debugging information, as it would increase the verbosity in such a way, that in testbeds executions the resulting data would eclipse the valued experiment data.

Listing 2.4: Declaring a callback method and registering it to the radio module.

```
1  //Definition of the receive method
2  void receive( node_id_t from, size_t len, block_data_t *data
       )
3  {
4       // Ignore if heard oneself's message.
5       if ( from == radio().id() )
6            return;
7
8       memcpy( &buffer_, data, len);
9
10      if ( buffer_[0] == MSG_1 ) action_msg_one(from, len,
           buffer_);
11      else if ( buffer_[0] == MSG_2 ) action_msg_two(from,
           len, buffer_);
12      else if ( buffer_[0] == MSG_3 ) action_msg_three(from,
           len, buffer_);
13      else if ( buffer_[0] == MSG_4 ) action_msg_four(from,
           len, buffer_);
14 }
15
16 //Registering it to the radio module
17 radio_callback_id_ = radio().template reg_recv_callback<
      self_type, &self_type::receive>( this );
```

Before departing from this receiving explanation, it is useful to introduce the delegate[1] system of the Wiselib. The delegates system is the main building block to the event driven system that embedded systems programming encourage. In event driven systems, instead of specifying what actions must be taken at which time and then spawn a periodic polling, the programmer sets actions that will be taken in response to events. This sort of lazy execution, which is only active as long as there are events to handle, is without any doubt beneficial to power and computationally constrained systems and

[1]For further detail into the delegates system of the Wiselib from an implementation perspective rather than the users perspective presented in the body of the text, I refer the reader to http://www.wiselib.org/wiki/delegates.

can maximize their possibilities. If, as stated, the delegates are the main building blocks to the event driven way of developing in the Wiselib, it is trivial to assume that their function is none other than to set the functions which will be called upon the occurrence of certain events. One example of that was presented in the previous paragraph when talking about radio's receive callback.

Listing 2.5: Using delegates

```
1  #include "util/delegates/delegate.hpp"
2
3  typedef delegate3<void, node_id_t, size_t, block_data_t*>
       sample_delegate_t;
4
5  sample_delegate_t sample_recv_callback_;
6  bool reg_callback = false;
7
8  /** Highway receive callback registering.
9  * @param obj_pnt An object with a method matching the
       receive signature.
10 */
11 template<class T, void (T::*TMethod)(node_id_t, size_t,
       block_data_t*)>
12 uint8_t sample_reg_recv_callback(T *obj_pnt) {
13   sample_recv_callback_ = sample_delegate_t::template
         from_method<T, TMethod > ( obj_pnt );
14   reg_callback_ = true;
15   return 0;
16 }
17
18 /** Highway receive callback unregistering.
19 */
20 void unreg_hwy_recv_callback() {
21   sample_recv_callback_ = sample_delegate_t();
22   reg_callback_ = false;
23 }
24
25 ...
```

```
26
27  if( reg_callback_ ) sample_recv_callback_(sender, len-
        ROUTING_OVERHEAD, &data[ROUTING_OVERHEAD]);
```

In the code snippet 2.5 the reader can see how to set interfaces for user defined delegates. To assist in the process, the Wiselib has a header with several prepared templates for delegates which one can reuse according to the amount of parameters his or her callback methods must receive on execution. On this example, we prepare a receive method, equal in interface to the one declared and defined by the radio module which could serve to encapsulate, for example, the sending and receiving from one end to the other of a wireless sensor network performed by a routing algorithm, which would only invoke the higher level receive on the target node, and not in each of the hops. On lines 3-6, the reader can see how we use the three parameter delegate defined in the included header, by feeding it, as template types, the types of the return data and the calling parameters. After that, and for convenience, as our hypothetic routing algorithm has only one possible callback method[1] we declare and initialize to false a boolean which indicates that no callback method has been registered yet. Once the variables are in place, the registering and unregistering methods follow suit with respective definitions on lines 11-16 and 20-23. Finally on line 27, the reader can find a sample call[2] to the stored callback method, which has a very natural syntax, thanks to the abstractions performed by the registering method and delegates header.

The next resource to talk about is the *Timer*. It is a module which serves a very frequent purpose in the scope of wireless sensor networks, and that is no other than to cover the need to execute a method with a certain delay or to periodically execute a method to, for example, check the amount of answering neighbors for connectivity purposes, the battery left, rebuild the routing tables, etc. The use of the timer module is relatively straightforward, as it basically needs the user to ask Wiselib for the timer

[1] Support for more complicated callback method mapping is easy to implement. As an example, one could declare a fixed size array of sample_delegate_t and store up to N methods to be called back, or declare a map which would only call a matching delegate on, for example, the event of the data parameter having a certain id.

[2] If the reader wonders what is the meaning of the stated overhead, it justs responds to the general case in which the data to route is encapsulated into a bigger routing packet (the extra size of which forms the overhead) which, on reception in the destination just returns the original packet and discards the encapsulating part.

facet, and then feed the time to wait in milliseconds, the object which has the callback method, and data to pass the callback method as arguments. If the user wants a periodic execution, he/she should set the timer at the end of the periodic to be method, thus making it call itself with a delay, achieving the intended purpose. It is important to note, that the callback method should have as its only parameter a void pointer. A trivial use could be:

Listing 2.6: Using the timer module

```
1 //Set a one second delay to the execution of the callback
     method.
2 timer_->set_timer<Application, &Application::callback_method
     >((millis_t) 1000, this, 0 );
```

A tightly related resource to the previous module is the *Clock*. The clock module, as the name implies gives access to the operating system clock and must implement at least, for each of the targeted platforms, a method called time which returns a time object. This time object can be processed by other methods offered by this module to be processed in a more humanly readable form such as seconds and milliseconds. The use of this module is similar to the others in the sense that one has to get the facet from the operating system. However, in contrast to the previously presented resources, this one has a mainly informative purpose, although it can prove very useful when, for example, deciding to discard packets which where travelling too long through the neighborhood, or timing the round trip (2.7) to reach a certain node. One such calculation could be:

Listing 2.7: Demonstrating the clock use by means of a round trip calculation.

```
1 uint32_t round_trip_in_millis = ( uint32_t ) clock().
     milliseconds( clock().time() ) + clock().seconds( clock()
     .time() ) * 1000 - ( uint32_t ) clock().milliseconds(
     sent_time ) - ( uint32_t ) clock().seconds( sent_time ) *
      1000;
```

To be able to collect metrics from the simulations and the testbed experiments, one must do so through extensive use of the *Debug* module. Through this module it is possible to print out messages with a similar syntax to that of the classic C *printf* command, i.e., specifying a template string and supplying the filling variables afterwards.

Obviously, such an interface can also be used in the debugging stages, albeit one should be cautious due to the fact that the standard Wiselib implementation uses the standard output, which is buffered in some platforms like the Shawn simulator, and thus, the last message to be printed can, in fact, not be the latest to be reached in the execution, and thus send the developer searching for ghost bugs. Fortunately, the Wiselib is open source, and it is trivial to swap the debug method to dump its writing to the standard error. Another particularity of this module is that the printouts can differ in the different platforms, e.g., surrounding the printed test in testbed information in the iSense executions, or displaying rounds information in Shawn simulations. To address these subtle complications, it is recommended to pick an easily parsable output format for your algorithms which can be processed later to a common format, regardless of where the execution took place. A simple call to the debugger would be as follows:

Listing 2.8: Demonstrating the debug use by means of a method exit printout

```
1 debug_->debug("%x METHOD_ENDED: foo()\n", radio().id() );
```

The last interface to talk about is the *Random* module. It can be useful in many circumstances like in calculating back off times for retransmission timers, or to solve draws in some hierarchical algorithms like for example deciding the leader of a cluster depending on the degree of connectivity. The use of the random module is rather simple, as seen in 2.9, one just has to plant the randomness seed, in this case the node's id and then specify the range.

Listing 2.9: Calculating a random integer modulus 100

```
1 random_->srand(radio_->id());
2 int num = (*random_)(100);
```

2.1.1.2 A quick overview of the pSTL

If the previous section was about processing data, this section is its natural complement as it deals with the standard way of storing the data to be stored, and that one is none other than using the pico STL, that is provided with the Wiselib, mostly in its testing branch. As introduced earlier in the text, the pSTL aims to mimic the standard C++ STL in its purpose and interface, but in a much more succinct way, i.e., centering in

the core data structures and providing the smallest implementations both in code size and in memory footprint.

Contrary to the library from which the pSTL draws its inspiration from, until very recently, there was virtually no support for using any kind of standardized dynamic data structures. Fortunately though, this is being heavily worked on, and will be soon part of the pSTL (some initial dynamic data structures can already be found in the testing branch), although in the current section we will deal with the more tried and reliable data structures that have been longer with the wiselib.

Data structure	Description
iterator	Definition of the iterator class, to be used in the definition of all the iterators in iterable data structures.
vector_static	Fixed size and iterable array based data structure. Elements can only be added at the end.
pair	Basic tuple of size two.
map_static_vector	Fixed size and iterable and array based data structured that is indexed by keys.
list_static	Fixed size and iterable data structure which allows for insertion in elements in any position.
priority_queue	Fixed size data structure which orders the inserted values by the "<" operator. Only allows to check the top element.

Table 2.3: Table of the main pSTL data structures

From the 2.3 table, it is quite clear what each data structure is supposed to do. However, there are some things that are not completely intuitive in their operation, mainly because nowadays we are much more used to dynamic size containers and some of the compromises taken to limit the size might not be completely straightforward to guess. The solution to adding something to full storage is to silently fail, i.e., not to add the element to the container and to return as if it had been done.Thus, it is advisable to check the capacity before inserting. The priority_queue follows a similar principle, when one would probably expect that when pushing a smaller element than the queue's worst, the latter would be flushed. To achieve a displacing push it is recommended to

extend the class or to make helper functions flushing and refilling the queue. In the map_static_vector, requesting a key implicitly creates it blank, which means that if you check for an element which didn't fit or was not initialized into the data structure, you are gonna get a properly zeroed out structure like the one you are requesting, giving the false sensation that the data is valid and it makes for a very hard to catch bug. For this reason, it is recommended to perform some checks like in the 2.10 code snippet.

Listing 2.10: Iterating and checking if a key is set

```
1  for ( pit = sample_map_.begin(); pit != sample_map_.end();
       ++pit )
2  {
3        //Check if the key exists.
4        //The key, value is saved as a pair, and thus accessed
            with the first, second keywords respectively.
5        hit = target_table_.find( pit->first );
6        // If hit equals the end of the map it means that it
            was not found, as the end element is reserved.
7        if( hit != target_table_.end() )
8        {
9             continue;
10       }
11 }
```

The research done in the process of designing and building the Wiselib has been awarded several publications in international conferences [13][14][15][16].

2.2 Shawn

From the previous sections, it has been emphasized the capital importance that testing has for embedded solutions development, for the obvious economic and time reasons that late cycle testing is much more cumbersome and risky. Wireless sensor networks are not an exception, but a clear example of that. Thus, an error or bug in the problem analysis could be fixed easily on the paper but, if it goes unnoticed onto the coding stage, it can cause the time consumption to fix the bug to be between one and two

orders of magnitude higher, growing the closer we are to production[1], i.e., in the real world experiments. Thus, each step down the following list, represents an incremental cost to find bugs.

- Analytical methods.
- Computer simulations.
- Real world experiments.

The first step, the analytical phase is generally done on paper and its aim is to match the problem with a skeleton of a solution in the form of a wire framed algorithm. However, Wireless Sensor networks are naturally complex, if only because the amount of data and study that has gone into it is relatively tiny compared to other computer science branches. This complexity makes the match between the wire frames and the final real world application a tough and hard problem. Thankfully, to ease the transition we can count on software aids like the simulators, which present a middle point in both detail and cost to fix the bugs. In the research presented by this paper, the simulator picked was Shawn.

Shawn is a wireless sensor networks simulator implemented in C++ that has its own application development model or framework based on processors, i.e., programs that process the inputs and generate outputs in rounds that conform experiments. The advantage of using Shawn is that not only it allows for specifying a lot of the simulation aspects such as duration, number of rounds, topology of the nodes and amount of nodes, but it is also directly usable with Wiselib applications by means of a generic processor that is distributed with the Wiselib[2]. This direct usability is almost transparent to the user, in terms that once the environment variables are properly set in Wiselib's Makefiles and all the dependencies covered, a simple *make* generates an executable which can be fed a Shawn configuration file to perform the simulation. This reduces a lot the complexity of exporting the Wiselib code to a Shawn processor and reduces the development time in a very invaluable amount of time.

[1]For a good description an generalization of this issue with extensive data to back it (although coming not just from embedded background, read [17, p. 28-33].

[2]The Wiselib Shawn application can be found in the repository inside the directory trunk-/Shawn_apps/wiselib.

In the following two subsections, the reader can find a generic description of the Shawn simulator, followed by a description of how it was used in the development of the Highways.

2.2.1 Description of the simulation principles of Shawn

Shawn design goals spring from its algorithmic background, as opposed to some other simulators which are more hardware oriented and center a lot of their efforts on the specifics on the communication. This fact makes it a simulator which is well aligned to the needs and aims of our research project. The official goals, as shown in the wiki page of the Shawn project[1], are:

- Simulate the effect caused by a phenomenon, not the phenomenon itself.
- Scalability and support for extremely large networks.
- Free choice of the implementation model.

Starting with the first of the previous bullet points, it means that what the Shawn simulator aims to do is let the developer gauge how different phenomenons, such as physical obstacles will affect a certain programmed network model, rather than center the attention of the developer on the physical aspects of the phenomena. This means, that the research is more focused on effects and impacts than otherwise, which makes this quite suitable for this kind of engineering research, which wants to explore the software, as opposed to do models of phenomena of wireless sensor networks. Of course, not everything are good news, as this deemphasizing of the phenomena will surely translate into a wider gap between the observations in the real hardware experiments and the simulations, but I consider that the faster algorithm modelling outweighs the detrimental effects of the initial discrepancies when taking a simulation validated module to the real sensors.

Larger network support is achieved by the slender network model taken by Shawn, which takes shape as a number of compromises or simplifications in the modelling of the networks. The simplification of several of the low level specificities than are naturally costly in computational terms allows the performance to be orders of magnitude faster for an equally sized simulation than a low level specific solution, or rather take on orders

[1] Further information about Shawn can be found at Shawn's wiki[6]

of magnitude bigger problems with the same amount of resources. Usually the latter version of the performance versus size is taken, as the FRONTS project aims to model wireless sensor networks in large societies, which without the Shawn simplifications would be really costly to simulate.

The freedom in implementation model is central to our choice of Shawn as our simulation software, as it is precisely this that enables us to run

2.2.2 Shawn as a testing, debugging and validation tool

Testing in the Shawn simulator is performed, in the case of Wiselib applications, through direct invocation of the Wiselib compiled binary, passing it an argument pointing to the Shawn configuration file, as follows:

Listing 2.11: Invoking a Wiselib application Shawn simulation

```
1  $ make shawn
2  $ ./highway_app -f options.conf
3
4  ------- Sample content of options.conf --------
5  random_seed action=create filename=.rseed
6
7  prepare_world edge_model=list comm_model=disk_graph range
       =3.5 \
8             transm_model=stats_chain
9  chain_transm_model name=reliable
10
11 rect_world width=8 height=8 count=800 processors=wiselib
12
13 simulation max_iterations=25
14
15 dump_transmission_stats
```

In the listing 2.11 the reader can see how by in two very simple steps, compiling the Wiselib application with Shawn as a target, and immediately running it with a configuration file, where one can set different scenarios to test the algorithm/application on. One of the most relevant options are the random seed to be used for placing the nodes in the map. The default setting is to generate a different placement on every

run, but this can be altered by specifying a different random seed action, namely *set seed=123456789*.

Establishing a static seed with the *set seed* command is a key advantage when tweaking the algorithm because it allows you to test every time on the same conditions, thus eliminating every variable but your latest changes to the source code, from the list of causes to the changes in the simulation outcome. Other interesting attributes to set in the aforementioned options file are the size of the simulated world, the number of iterations to run (where every round equals to roughly a second) and the transmission range, in units of distance, of the individual nodes.

The output from the execution of the simulator with a similar configuration file as the one listed previously consists of a structure such as the one found in the listing 2.12. In it one can see how Shawn initializes the simulation engine according to the specified parameters and shows the debug messages of each of the nodes organized in rounds enclosed by *BEGIN ITERATION N* and *DONE ITERATION N*. In the end, another very useful piece of information is the dump of transmission stats for the simulation session. In them, we can get an idea of our application's use of the transmission medium, the connectivity, etc. This connectivity allows us to get an idea of how well the network layer is performing by comparison.

Listing 2.12: Output from a Shawn simulation

```
1  init_apps: init_routing(sc); init_localization(sc);
       init_external_application(sc); init_reading(sc);
       init_topology(sc); init_examples(sc);
2  Initialising External_Application  Shawn module
3  Initialising Wiselib-Shawn-Standalone module
4  init_topology_elevation
5  init_topology_generator
6  init_topology_node_gen
7  init_topology_node_mod
8  init_topology_point_gen
9  init_topology_point_mod
10 init_topology_topology
11 Initialising examples
12 Initialising Wiselib-Shawn module
13 Simulation: Running task 'random_seed'
```

```
Simulation: Task done 'random_seed'
Simulation: Running task 'prepare_world'
DiskGraphModel: Transmission range set to [300]
Simulation: Task done 'prepare_world'
Simulation: Running task 'chain_transm_model'
Simulation: Task done 'chain_transm_model'
Simulation: Running task 'rect_world'
Simulation: Task done 'rect_world'
Simulation: Running task 'simulation'
------------------------------ BEGIN ITERATION 0
SEND JOIN [11 , 0 , 0]
SEND JOIN [11 , 10 , 0]
SEND JOIN [11 , 20 , 0]
...
------------------------------ DONE ITERATION 0
  [ 100 active, 0 sleeping, 0 inactive ]

------------------------------ BEGIN ITERATION 1
...

------------------------------ DONE ITERATION 1500
  [ 100 active, 0 sleeping, 0 inactive ]

Simulation: Task done 'simulation'
Simulation: Running task 'dump_transmission_stats'
---- stats_chain transmission model information
   -----------------
general all_types messages 151203
general all_types size 14544003
general N7wiselib14WiselibMessageIihlEE messages 151203
general N7wiselib14WiselibMessageIihlEE size 14544003
-----------------------------------------------------------------

Simulation: Task done 'dump_transmission_stats'
Simulation: Running task 'connectivity'
tasks.connectivity: Mean connectivity: 21.98
```

```
48 tasks.connectivity: Min connectivity: 8
49 tasks.connectivity: Max connectivity: 35
50 Simulation: Task done 'connectivity'
```

As the reader can see in the previous listing, the spaces that the developer has for generating parsable debug messages[1] are defined by enclosing rounds definitions. This is a useful situation compared to the real hardware experimentation (which usually arrives out of order) as it allows a clear way to follow the flow of the application.

Furthermore, Shawn incorporates a visualization module with support for *live* rendering of the evolution of the network which has been used by some developers of the FRONTS project to produce very descriptive snapshots of the system. This module, however, was dismissed from this project due to portability concerns, as it is not available on the most important experimentation part of the project (the real hardware experiments). To substitute such a useful feature in a portable way, a couple of visors[2] of increasing complexity and capabilities were developed in Python during the scope of this project.

The research involved in the design and implementation of the Shawn simulator has been worthy of publication in several international conferences [18][19].

[1] To generate the graphics and metrics that will be presented in the experimentation chapter of this thesis, the approach followed was putting specially formatted messages on Wiselib debug methods and writing several tools in Python and BASH to parse them and generate metrics and snapshots of the algorithm performance

[2] A description and samples of the input and output of these tools can be found on the tools chapter

2.3 iSense Hardware

Figure 2.2: The core module of an iSense sensor

As stated in the section 1.3 (Highway construction), more precisely on the page 11, the bulk of the experimentation and decisions that shaped this document and the algorithm that it describes, were taken with real hardware on mind in most of the situations. The real hardware that is referenced throughout the document is none other than the product for wireless sensor networks called iSense, which is manufactured and distributed by the German company Coalesenses GmbH.

The iSense hardware describes rather a platform than a device, as it is composed of a base or core module, such as the one shown in figure 2.2 with UART connectible ports that enable the attachment of additional boards (seen in figure 2.3) that enable different sets of features, depending on the nature of the network to be deployed. Some examples of these pluggable modules are the environmental and the solar ones, which enable for the construction of a compilation of data on buildings conditions such as the one made available by Barcelona Tech in the scope of the WISEBED project[1].

In regards to the core module, which the reader can see at the center of the figure 2.3, it is composed of a 32-bit RISC jennic microcrontroller[20] with 128Kb of ROM and 92Kb of RAM. Embedded on the microcontroller there is also a 802.15.4 3dB/-97dB radio module with support for encryption. The programming language targeted by the microcontroller's maker compiler is C++, which falls right in line with the programming language of choice of the Wiselib. It's easy to see, that the memory constraints of the platform are not to be taken lightly, and even more so when one takes into consideration that the platform pulls the code into the ram on power on to fetch the instructions from there. Not using an instruction fetcher hooked to the EEPROM makes the available RAM for the iSense Operating System and the applications rather small, and in practice, code sizes of more than 85Kb were found to have memory issues.

[1]This data compilation and similar ones obtained from the sensors deployed at Barcelona Tech can be found in the WISEBED defined WiseML format at http://albcom.lsi.upc.edu/fronts/?cmd=solar

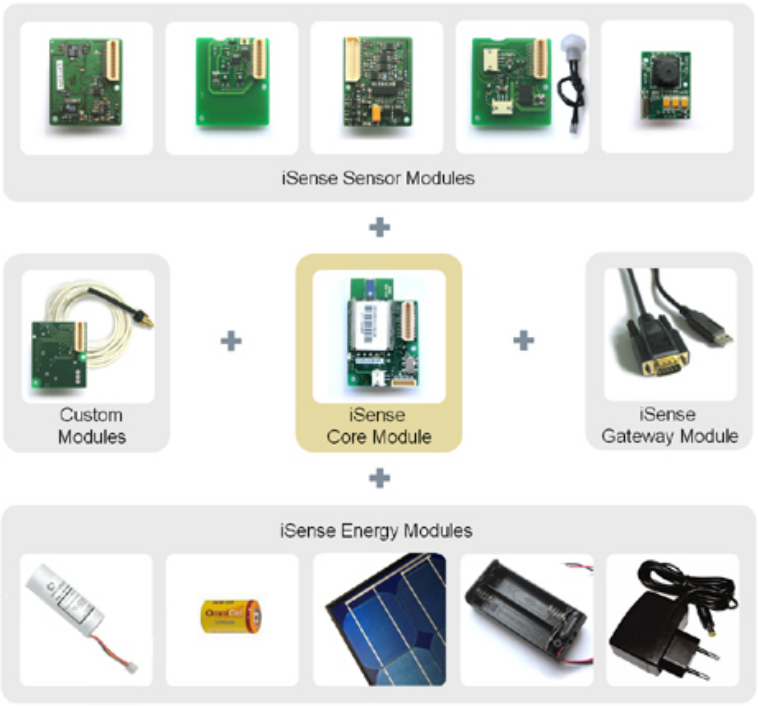


Figure 2.3: iSense core and modules architecture

The iSense Operating System is a proprietary software developed by Coalesenses GmBH, and it provides access to the iSense hardware modules and the Jennic microcontroller features through a C++ API. It is quite possible and extended to develop applications straightly in the Coalesenses provided iSense ecosystem. The Highway module though, interfaces only with the iSense Operating System through the Wiselib, which acts as a proxy of the system calls of iSense. It is important to mention, connecting the Operating System interface to the hardware and the memory size constraints of the platform, that just as the hardware is modular, so is the operating system. The modularity is practically made visible and useful to the developer by a firmware baking Coalesenses webservice[1] of the that lets the developer pick which features, modules, buffer sizes, etc. to build into the firmware, that will later be loaded into the iSense sensors.

The development[2] work flow of the iSense ecosystem, not considering the testbeds

[1] In the following URL, the reader can select an Operating System revision and its configuration http://www.coalesenses.com/index.php?page=webcompile

[2] The development environment tools work both in Microsoft Windows and Linux platforms and

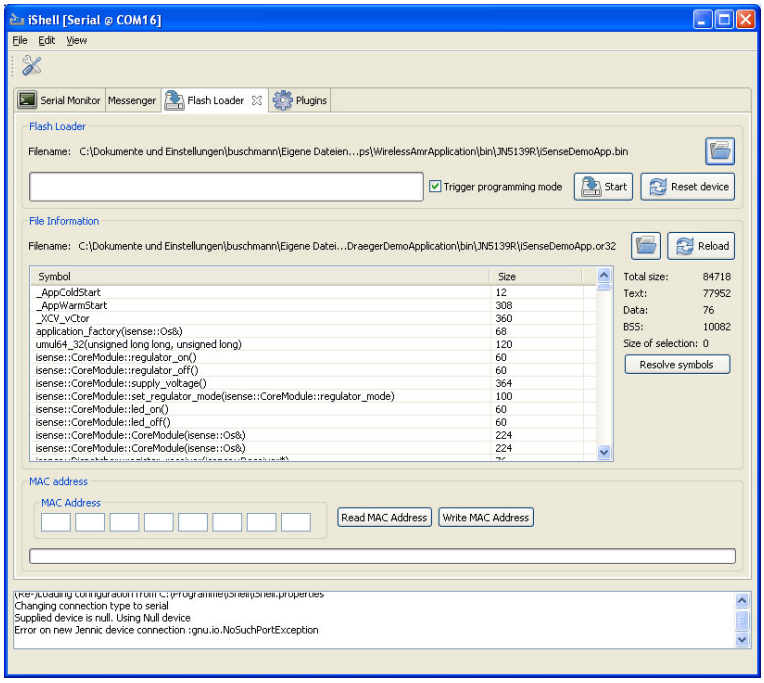


Figure 2.4: iShell application for loading and running iSense applications

but focusing on a small amount of nodes connected to the development computer, consists on coding the application, compiling it and performing the linking to the Operating System/firmware configured through the webcompile, and loading the application to one or more iSense devices through a custom Coalesenses tool called iShell, the main window of which is depicted in figure **??**. Through iShell, not only does one get to load the binary containing the application, the Wiselib and the iSense OS, but also get access to the debug messages issued by the code during its execution (it is possible to monitor several iSense devices through tabs on the application). The communication between iShell and the iSense devices is conducted through the iSense gateway module, which is basically composed of USB to RS-232 gate to the microntroller used for data exchange and powering of the device.

Even though it is very useful to develop and test Wireless Sensor Network applications in a few iSense devices connected to the development computer, especially during the initial stages of the real hardware tests, when one starts to hit the walls of the platform constraints, it is of limited service when testing algorithms that are meant for

are available at http://www.coalesenses.com/index.php?page=development-environent

large scale societies of nodes set apart by a lengthy distance. In the next section we will introduce the testing facilities in which the real experiments were conducted.

2.4 iWSN testbed software

Wireless sensor networks systems, as per the FRONTS definition, should be composed by a large adaptive body of inexpensive devices, and taking the iSense nodes as devices, a way of interfacing with big amounts of them was needed in order to research the new algorithms and computing models. To address this issue and other research targets, the WISEBED project designed and implemented the iWSN testbed software.

The iWSN testbed is a Java software solution that is made available by the Universität zu Lübeck and used by all the WISEBED project testbeds. The architecture of the system is a partially distributed software that is structured in controller and base stations. The controller is the endpoint of the testbed communication with the clients, from whom it receives the reserves to use the testbed and the software to load into the iSense devices. The reservation system is backed by a MySQL database and features a very useful integration with Google Calendar[1], which allows the researchers in the different sites to get an idea of the availability of the testbed to better schedule experiments. The controller interfaces with a java client, which is part of this same solution. The client can be tweaked via Beanshell Scripting enabling modifications such as changing the experiment time, rounds, presentation of the retrieved data, authentication, etc.

The client part of the iWSN testbed software is not just a client, but a dual client/server solution. The client part is in charge of scheduling an experiment with the testbed by authenticating to it, requesting a time window, setting the experiment parameters and submitting the binary file to be loaded on the iSense devices that it chooses to run the experiment on. The server part of the client is set up to listen for a connection from the testbed which will feed back, almost in real time, the debugging messages generated by the picked iSense devices. The debugging messages are collected by the testbed software as they arrive from the different base stations and are streamed to the client's server port in an order which is not guaranteed to be the one in which they were

[1] A very good example of the feature is the busiest testbed Google calendar page https://www.google.com/calendar/b/0/embed?src=testbed-itm@itm.uni-luebeck.de&ctz=Europe/Berlin&gsessionid=OK

generated. The client's server part then outputs the stream of debugging messages to the console by default.

As getting the experiment on the standard output is generally not convenient, because one normally want to extract multiple kinds of information from a sole experiment run, it is more useful to write a small BASH script that classifies the experiments on a results folder according to which Beanshell script and which run number originated it. In the code snippet 2.13 the reader can see such that automatically redirects the output to a named experiment. The listed script will take the name of the Beanshell script, that serves as experiment identifier and will attach to it a number obtained from increasing by one the last experiment found in the results directory with the same experiment id. Moreover, it gives feedback to the user regarding the placement of the file for extra convenience.

Listing 2.13: BASH script for running the testbed client

```
1  #!/bin/bash
2  # To run this script just set its permissions to executable
       and run it with the beanshell
3  # script as its only parameter. For example: ./run.sh
       highway_luebeck.bsh
4
5  # Parameter fetching
6  NAME=‘echo $1 | sed 's/\..*//'‘
7  BEANSHELL=$1
8
9  # Getting experiment number by id.
10 NUM=‘ls -l results/$NAME* 2> /dev/null | wc -l‘
11
12 # Generate the testbed client command
13 CMD="java -jar tr.wisebed-cmdline-client-0.6.1-onejar.jar -f
       ${BEANSHELL} -v"
14
15 # Run it and leave it on its rightful place
16 echo "Saving results of the experiment to results/${NAME}
       _$NUM.txt"
17 $CMD > results/${NAME}_$NUM.txt
```

In the testbed server side, the controller starts up the other systems that take part in the testbed, also known as base stations, by using parallel ssh, a secure way to multicast commands to all the machines defined in the testbed configuration files. This parallel commanding system is extensively used by the testbed administration tools, especially worth mentioning the use cases of deploying new updates to the testbed runtime and for stopping and restarting it. To achieve safe and password less administration, the testbeds relies on the RSA asymmetric key encryption provided by the Open Secure shells.

On the lowest level, the base station software uses a mostly LGPL[1] Java library called RXTX[21] which is a native software solution wrapped around for calling from the virtual machine. The functionality that this library provides is serial port communication, which is used to load the firmware and to collect the data from the experiments.

[1]The lesser General Public License is a Free Software license. More information about it can be found at http://www.gnu.org/licenses/lgpl-2.1.html

3

Algorithm design

The algorithm that is subject to this chapter is designed with the purpose of performing highway construction in wireless sensor networks. A highway is a communication path built upon hierarchical structures formed by the association of nodes into clusters with an elected leader.

For the construction of highways in wireless sensor networks we must have a series of mechanisms to allow higher level treatment of clusters as single entities, i.e., virtual nodes. This change in the clusters treatment means that the cluster leader becomes the logical leading entity of the virtual node, and the leader, but mainly the rest of the cluster nodes, become the communication ports of the virtual node with their neighboring virtual nodes. The list of nodes that constitute communication paths between leaders and are chosen by both leaders to perform this task is what we call highways.

The hierarchy construction is performed in collaboration with the neighborhood discovery and the clustering algorithms, developed at the CTI. Both algorithms play the crucial job of being the senses that generate the events to which the Highway algorithm reacts. This is a tremendous responsibility as the nature of the algorithm presented here is purely event driven and thus, the success or failure is greatly dependant on them.

In this chapter, the reader will find a description of the problems and challenges that the author addressed with the work behind this thesis, as well as a description of the proposed solution to it. The description will be further detailed in the next chapter, where the reader will be presented with the specifics of the implementation of what is described in section 3.2.

3.1 Problem description

The problem that this thesis addresses is that of establishing a hierarchical routing on wireless sensor networks. The definition of this kind of routing is building or finding paths within the wireless network that connect the already established hierarchical groups, also known as clusters. It is important to note, that we also set as a problem to solve the way in which this path finding should be achieved, and this is none other than to perform the highway construction in the most distributed way possible while maintaining the logical leaders of the clusters in control.

wireless sensor networks present a lot of challenges, as the reader must have noticed from the introductory words about the FRONTS and the WISEBED project. Thus, the problem that was set before us in the beginning of this project is not just the one mentioned in the previous chapter, which has a very theoretical sound to it, but a whole array of side problems and complications that come from performing research in such a relatively unexplored field.

The first of these side problems is that highway construction cannot be built upon the assumption that the point to point communication between nodes of the network is reliable. This fact has itself very deep implications for the whole networking stack, as it means that fault tolerance methods must be deployed throughout the whole layer. This translates into the added complexity of dealing with the adverse effects that are carried up the layer us problems appear on the lowest level. If fault tolerance were not adequate in the lower layers, from which the higher level modules get all their information, the up propagated data could diverge more and more from the real situation and mislead the top layers to react on the base of inexistent scenarios.

Another implication of the unreliability of the network is that changes can occur often enough and revert to the previous state that sometimes reacting to an event can be utterly counter productive, to the point of unnecessarily disconnecting a part of the network, were perhaps it would have been better to delay the decision, even when that would mean a more ad-hoc approach to the problem and a sacrifice of algorithmic cleanliness. For this reason and the error propagation stated in the previous propagation, it is important to treat received information inspection as an added problem to address.

A very important problem when dealing with real life implementations is the level of knowledge of the setup, materials and scenarios in which the engineering process has

to take place. This project was not an exception to this principle, and the problem of learning the technologies of choice of the FRONTS project while marching ahead towards a looming deadline that had to be met using those precise technologies was certainly challenging. It is obvious that this is to be expected when working on young and rapidly evolving technologies such as the ones described in the previous chapter. This is not to say that they were problematic, but it is undeniable that stomping into unsuspected constraints without a clear knowledge of their origins adds a great deal of complexity to the problem solving process.

The last of the main side problems or challenges to be mentioned in this section is the most human of them all, the problem of communication of geographically distant teams. From reading the previous paragraphs, the reader can rapidly see that most, if not all, of the issues affecting this party of the project affect as well all the other sites involved. The fact that there is a high degree of interdependence between to the modules only makes this matter graver, as the effects got multiplied by the rising issues being often solved by several parties individually instead of addressing them as a team. Undoubtedly, the busy and different schedules of all the parties involved did nothing to change this fact. However, on a positive note, the workshops progressively decreased the impact of this side problem as well as many others.

Leaving the side problems behind and shifting focus to the self-standing problems that the next section deals with, the author believes that a down to top approach is appropriate and as such, let's start with the neighborhood discovery.

Regarding the neighborhood discovery, the problem is to determine an efficient way in which to gather data from the network with which the clusters can find alien nodes that can potentially act as a gateway for the hierarchical routing that we aim to perform. It is important to take into consideration in this step two of the classical concerns in wireless sensor networks, namely power consumption and medium usage. Solving the neighborhood discovery problem, from the perspective of the Highway algorithm means finding a way in which both concerns are minimized. Thus, the algorithm must be restrictive with the amount of communications and or with their strength and length. Since the neighborhood discovery module controls the first parameter, the problem is best dealt with by minimizing the data to be piggybacked to the neighborhood discovery beacons.

In the clustering level, the problem is to react adequately to the instability produced by the fluctuations in the network that cause changes in the groups that compose the network. Adequacy, in this case, means to be able to build and destroy highways in response to the information relied by the network, while at the same time keeping the power and medium considerations. Being to eager to change could mean unnecessarily short lived highways and a network flooded by highway establishment packets.

Already in the highway level, the problems that we face are of a similar inspiration as the ones announced on the previous two paragraphs, but with a new twist, that we must also consider a new requirement, the delivery rate. This means that that the job of our module is to build a functional layer on top of mainly informational layers, and this implies that the problem of reliably encapsulating the data from a cluster to another is of central importance. Naturally, the dynamic nature of the wireless sensor networks and the power concerns pose a threat against the success of the functional part of the Highway algorithm. A part from being a serious threat they also force the establishment of trade-offs between on one side delivery rate and in the other reflecting the latest changes and consuming the least amount of energy while doing so.

3.2 The Highway module

The Highway module consists, as the reader might have gathered from the quite telling description of the problems to address, in using and pondering the data received from the two underlying modules and using that data to find paths that interconnect clusters. In this sense, one could think that the relationship with both, neighborhood discovery and clustering, is a completely passive one, but this is not completely the case, as their own design demands different degrees of involvement.

3.2.1 Sensory input. Event generation

Doing again a bottom to top review, we start with the neighborhood discovery module. Its architecture consists in broadcasting beacons, in fixed time intervals, to all the nodes within reach, and inferring a topology or rather a connectivity table from this process. It allows the overlying modules to register callbacks to the different events that it produces, as well as to reserve a certain amount of bytes in the beacons for piggybacking information. The available events are:

- *NEW_NB*: New neighbor added to the neighbors table.
- *NEW_NB_BIDI*: New neighbor that has bidirectional connectivity with the current node.
- *DROPPED_NB*: Complete loss of connectivity with a neighbor.
- *NEW_PAYLOAD_BIDI*: Received a beacon.
- *LOST_NB_BIDI*: Loss of the bidirectional property of a link with a neighbor.

As the Highway algorithm aims to find and build communication with the maximum efficiency, from the previous events, we take an Hard In Hard Out approach, which translates to registering only the *NEW_PAYLOAD_BIDI* event and optionally, the *DROPPED_NB*. The former is the most restrictive of the events that imply a new connection because not only it demands to receive data, but it demands that both nodes have each other in the neighbors tables. The latter is only registered in some of the optional implementations of the algorithm for achieving a faster propagation of a highway breakdown, albeit at the expense of energy and medium occupation. So, as announced, the algorithm it is relatively restrictive in incorporating nodes to the process and doesn't remove them unless there is a major reason to do so (which will be explained in a few paragraphs).

Omitting the dropping events has two contrasting consequences, the first one is that, in this way, the algorithm is less prone to restart the path finding routine (in the case that the "fallen" node was part of a highway) on a short lived interruption of the communication with a node that could be due to, for example, some RF jamming. On the other hand, it delays the adaptation to a probable change in the topology of the network that could potentially result in a decrease of the delivery rate. The justification of the choice of the former criterion over the latter is that this approach is less expensive and, as the reader will see in detail shortly, the quickness of the readjustments is bound by the clustering because highways perform inter-cluster routing. For this reason, we defer the removal of nodes to the clustering events.

The clustering module, which is the one that influences the Highway algorithm the most, groups the nodes according to several criteria, trying to maximize the proximity of the nodes that decide to be in the same cluster by limiting the response time. The events that the clustering module generate are more passive than the ones described in

the neighborhood discovery, as they do not allow for piggybacking. In the following list, the reader can see which are the available events to register to the clustering module:

- *NODE_JOINED*: The node has joined a new cluster.
- *CLUSTER_HEAD_CHANGED*: The node became cluster leader.
- *CLUSTER_FORMED*: Some nodes joined the cluster.

Where the neighborhood discovery generates the connectivity events, the clustering generates what we could call resetting events, as they are basically used to detect when a node switches clusters or it simply becomes a mono-node cluster. To ensure the change, the highway algorithm saves the previous cluster leader, and does not perform any resetting of the node unless the new event is setting a new leader, case in which, apart from resetting the node, in some implementations of the highways, if the current node used to be the cluster leader, tries to notify the other cluster leaders of the change.

3.2.2 Basic principles

The basic principle of the Highway algorithm is to treat the clusters, as a whole, as one big node, or logical node. In this virtual node, the cluster leader performs the tasks of coordinator for as long as they maintain leadership, and the rest of the nodes do communicating tasks. Due to the fact that, to save memory, the cluster leader and the virtual node or cluster that they command share the name, one could say that the gregarious nodes are just, from the Highway algorithm point of view, far reaching antennae. From the Wireless Sensor Network point of view, though, being used as communication extenders does not disable them for performing data gathering through their sensors.

The logical node formed by the cluster is a mutable entity, i.e., it can add and lose members while preserving, where possible, its paths to other clusters and its identity. The only exception is, understandably, when the leader leaves the cluster as, at least, the remaining nodes will not be able to be identified by the lost leader. However, the issue is far from being just a naming scheme, it has deeper implications, the most important of which is the fact that it is very common for the clusters to split and or be absorbed by their neighboring clusters. Another consideration is that, even if the nodes managed to stay together (minus the lost leader), all the internal pathways would have

to be routed anyway to the leader, making much less attractive any possible advantage over starting with a clean slate.

In terms of communication capabilities, the cluster leader's edge over the other members of the logical node is basically the ability to encapsulate and send data which will arrive to the destination cluster leader without any intervention by the overlying modules (typically the end to end module). In all other connectivity concerns they are to be considered the same as their fellow members of the cluster, including being able to send directly a message to another cluster. To do this, a tree structure is generated to be able to route the packets outwards and a map is used at the edges of the cluster to know to which external node to pass the information to.

The tree structure is only visible at the virtual level, because it is not saved, as a whole, in any of the nodes. The storage, when looking at the individuals that form a cluster, takes the shape of maps for the children and a one-to-one relationship with the parent. This simplification is a clear trade-off between inspecting potential of the individual nodes to take more informed decisions, and memory and code size consumption. Needless to say, our solution picks the latter option but tries to minimize the detrimental consequences of the decision by devising a collaborative path finding that reduces the impact of the data distribution.

To explain the data structure population process of the implicit tree, first we have to go into more detail as to how the neighboring virtual nodes or clusters are detected. When a node is notified that it belongs to a certain cluster (including becoming itself the cluster leader), it starts to send a beacon with the identification of its home cluster. Close by, there will likely be nodes of another cluster broadcasting their own collective identification. In the event that two or more of these nodes are able to exchange the information of to whom they belong, they will start a time window for listening to other communication opportunities. After this time window, they will send a message to notify the cluster cluster leader the finding of the neighboring cluster. This message is considered by the leader to be a candidacy to become a port of communication towards another cluster.

The port candidacy travels up the tree to the parent using the one-to-one relationship attribute provided by the clustering module in regards of who is the immediate parent of a node. In this route to the leader of the cluster the routing maps of each

node are updated. As the reader can see, these maps will only be updated with information regarding reach nodes with potential to be included in paths for outbound communications. The selectivity of the map updating aids in the memory saving and helps to make lookups faster[1], while not suffering any sizable penalty from the lack of exhaustive information.

3.2.3 Highway negotiation

Once the candidacy reaches its target, i.e., the cluster leader, a time window will be started in which more candidacies are allowed to enter the process that will ensue. In the case that the candidacy originated in the cluster leader, the candidacy window starts immediately. On the end of the time window, the cluster leader takes goes through the list of external clusters for which it has received a candidacy, and it issues a port request to them through a specific node of the cluster. The port requests are identified by their four main entities, namely, the cluster leader that originates the request, the node that issued the candidacy, the node the beacon of which translated in to the candidacy, and the cluster leader of the beaconing node. It isn't hard to see, that these are not the only nodes that can intervene in the negotiation of a highway, but they are certainly the ones that make or break the communication.

The choice of which node, of all of those who registered a candidacy during a time window, to pick to carry the port request to another cluster largely depends on the implementation, and although the reader can see that there are only two implementations (sections 4.1 and 4.2) detailed in this document, it is important to list all of those that were considered during the design and prototyping of the Highway algorithm.

- *LIFO*: Last In First Out, the port candidate that is picked for each cluster is the one which sent the last candidacy to reach that cluster.
- *FIFO*: First In First Out, the port candidate that sent the first candidacy will also send the port request.
- *SIFO*: Shortest In First Out, the port candidacy with the lowest amount of announced hops is picked to request a highway.

[1] At the time of writing, maps in the Wiselib are implemented as arrays of pairs. The lookup of a key in this implementation is implemented as a sequential matching of the keys, and thus O(n).

- *MIFO*: Most insistent First Out, the port candidacy that has announced itself the bigger amount of times is picked to negotiate a highway.

The first and the second are generally easy to implement, as the data structure they need to implement is just a map indexed by the target cluster identifier and containing as a value the candidacy. In terms of control flow, they are also very uncomplicated as well, just one conditional statement and it is done.

The SIFO approach, demands an extra storage field, the amount of nodes in the would be highway specified in the candidacy. In terms of code, the code is relatively simple too, only needing to check on reception if the candidacy is the shortest received in the current time window.

The last of the picking algorithms considered, MIFO, just like the previous algorithm seems to demand an extra field. However, this is misleading, as multiple candidacies for the same highway are highly unlikely to occur in the short time a candidacy window is opened. This makes it necessary to create a data structure that keeps counting between windows, without polluting the window. In terms of code, it is also a higher cost solution, as it demands coordinating two data structures.

From the previous paragraphs, it is quite clear which of the algorithms passed from design to implementation. The first three were all implemented and tested, albeit SIFO was only tested in simulations due to iSense code size constraints.

The advantage of LIFO is that it results in using the most up to date information, which makes it a good choice for highly volatile networks. In the FIFO case, the advantage is that the mutability of the data structures is smaller and one could potentially shorten the time windows on certain conditions such as on filling a certain amount of target cluster slots. For SIFO, the main advantage is that on large hop clusters it can pick shorter paths, which can result in more reliable and faster communications. Finally, MIFO allows to consistently pick the most reliable nodes in a stable topology.

Due to the requirements of the wireless sensor networks available through WISEBED and the FRONTS modules that, as explained earlier in the chapter are the eyes and ears to the topology, the favoured algorithm was LIFO. The strengths that made it the algorithm of choice were a smaller memory footprint, which was a better fit for the iSense memory constraints and the fact that a notable amount of layers had to be fitted

in them as a goal of the FRONTS project, and the fact that the underlying modules presented a highly volatile or changing topology to the highway module.

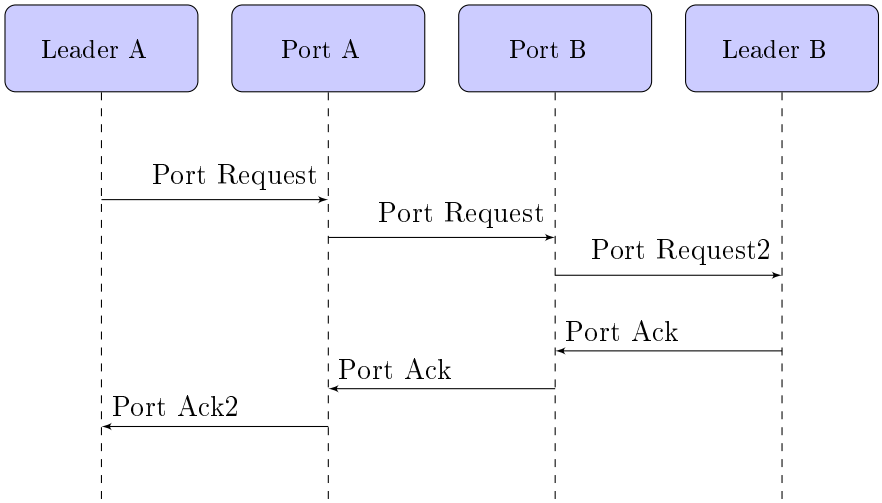
Back to the process of creating a highway, after picking a port, the kind of negotiation that was picked was a two way hand-shake. The obvious advantage for a two way handshake is the higher speed in which a highway can be set up, which also translates in a smaller number of transmissions, implying less power and communication medium usage, as well as less chance for packet loss. A sample of negotiation can be seen in Figure 3.1.

In figure 3.1a One can see how a successful negotiation looks like. The decision on where to send back an acknowledgement or a non acknowledgment was studied under the following policies:

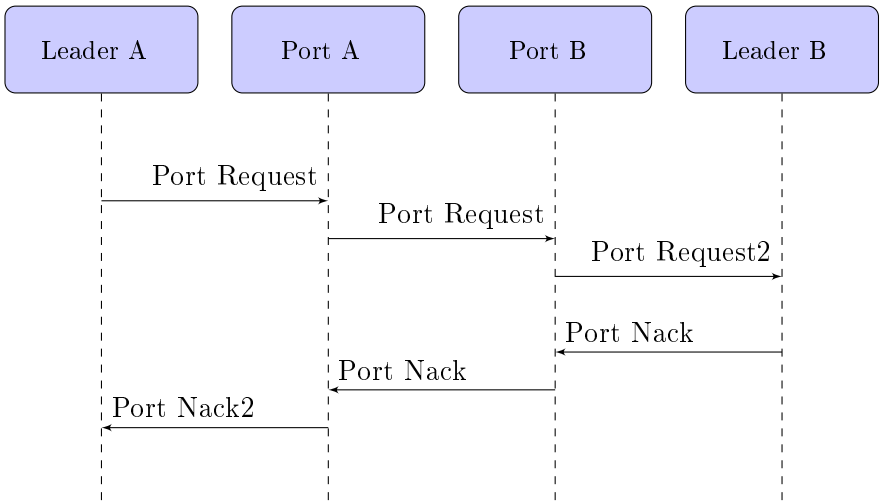
- Always acknowledge.
- Acknowledge if shorter.
- Acknowledge if no highway set, the requested is already set or current one is broken.

The always acknowledge policy follows the thought that, under a highly volatile network, it is always better to have the newest found path as the one used for communication. The second decision method works under the assumption that it is always better to traverse less nodes to achieve faster communication and to reduce the probability of losing the packets on the way. Finally the third, takes a more conservative approach and only establishes a new highway if there are no working highways. The condition stating that the highway is already set is because it could happen that one cluster is jammed for a period of time, as a consequence deems all the highways as broken, but when the communication jamming ends tries to get new highways, and asks another cluster for a path which the second cluster had not considered as broken.

During the testing of the highways it was shown that the second approach produced worse results, as reliable long paths were ignored in favor of shorter paths of nodes that were frequently swapping clusters. The first and the third approaches showed similar performance and were both used indistinctly with without any statistical difference. The lack of differentiation is probably due to the fact that the cluster A will not request a new highway unless it the current one is broken, reducing the always accept policy, in almost all the use cases, to the third policy.



(a) Successful highway negotiation



(b) Unsuccessful port negotiation

Figure 3.1: Complete two way handshake negotiations.

From the choice of acknowledgment versus non acknowledgement described above, the reader can see that it the unsuccessful negotiation depicted in figure 3.1b rarely happens. The most usual case of negotiation failure is when one of the edges of the diagram is broken by a communication mishap.

3.2.4 Highway maintenance

Once highways are established, they are made accessible to the higher layers by an interface called *cluster_neighbors*. This interface returns a list of virtual nodes that are currently attached to the current virtual node by means of working highways. To determine if the highways are functional, their maintenance system must be reviewed.

The design for maintenance of the highways consists in setting a maximum amount of non acknowledged transmissions. This implies that all for all the communications that a cluster leader receives from highways, it issues an empty acknowledgement message back to the sender. The empty acknowledgement message does not contain any reference to which message it is acknowledging, because that would imply extra transmitted data and transmission buffers that are too costly for a communication that is not expected to have a premium degree of reliability. The design of the acknowledgment tracking favors successful transmissions over failed ones, i.e., in the count of received acknowledgements, the successful case detracts a higher amount from the balance than the message sending increases. This design decision was put in place to be more tolerant during the stabilization phase of the network, in which the transmission failures are more common, and sticking with highways that are intermittently working reduces network overhead without an excessive penalty in delivery ratio.

When a highway exceeds the globally defined score of unacknowledged transmissions[1] it is removed from the highways map. Additionally, a last attempt of highway traversal is done with a non-acknowledgement packet with the intention of notifying the other end of the highway that it should be dropped from the highway map.

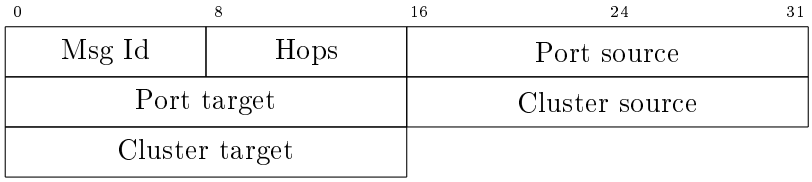
The choice of attempting to send back a notification to the other end of communication is a controversial one, as it spends energy to reach a node which logically should be unreachable (as it has been determined a broken path). A possible solution would be to make the notification in a multi hop broadcast transmission with a time to live set in the message. Interestingly enough, the tests show that the second approach is detrimental due to an increased amount of medium and power use, without achieving a significant advantage in successfully delivered notifications. The reason for this is that often an unfit highway is still navigable with a small probability that is not very far from the probability of a not very far reaching multi hop broadcast.

[1]Score is used instead of count due to the disassociation of the value and the received acknowledgements derived from the asymmetric increase decrease explained in the previous paragraph

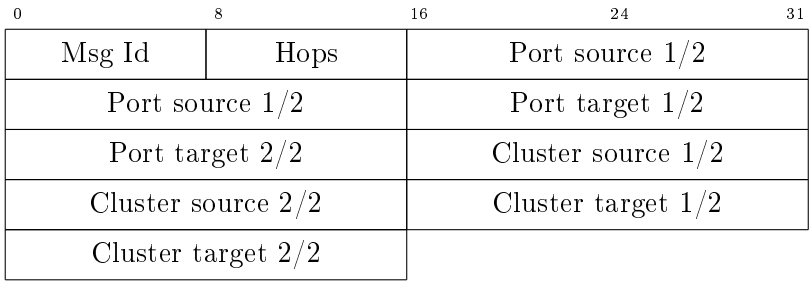
3.2.5 Highway transmission

The design of the highway transmission model is basically the definition of the packets which the highways use to transport their data, as well as establishing the communications.

Taking as main principles the maximum amount of code reuse and the aim to reduce the memory footprint of the application, the highways define, for all their operations just two type of messages. The first kind of message is the highway message, which is used for all the highway announcement and negotiation protocol described in the previous subsections. The second kind is the highway transport message, which consists of a large buffer in which to encapsulate and transport the information passed to the highway module for transmission.



(a) Highway iSense message packet



(b) Highway iSense message packet

Figure 3.2: Highway message packets.

In figure 3.2 one can see the design of the highway messages in both the iSense platform (figure 3.2a) and the Shawn platform (figure 3.2b). The reason for having two different designs is that the byte length of an id in both platforms differs. This is handled by the C++ preprocessor, but obviously requires having two preprocessed branches in all the code that deals with the packets. The explanation of the fields is pretty straight forward, as it consists basically of the message identifier, the distance of the highway to be, and the four players that determine a highway, namely cluster leader or origin, port of the cluster of origin, port of the target cluster and leader of

the target cluster. It is not hard to see that some pieces of the information could be dropped in some of the use cases, but that would force an amount of extra code that due to platform constraints the project could not afford.

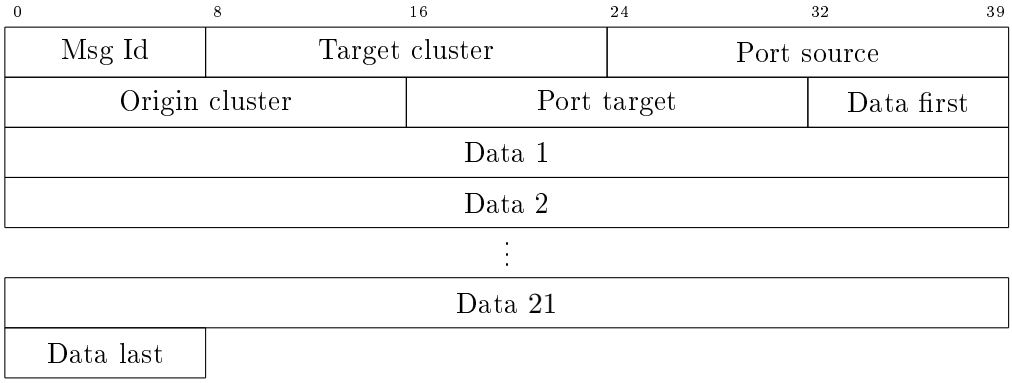
The messages that follow this structure are the following:

- *CANDIDACY*: Candidacy message issued after the discovery time window.
- *PORT_REQ*: Port request issued by the cluster leader after the candidacies time window.
- *PORT_REQ2*: Port request as it travels up the virtual tree of the target cluster.
- *PORT_ACK*: Port acknowledgment issued by the target cluster leader as a response to a port request.
- *PORT_ACK2*: Port acknowledgment as it travels up the virtual tree of the source cluster.
- *PORT_NACK*: Port non acknowledgement as issued by a cluster leader to notify a highway deletion.
- *PORT_NACK2*: Port non acknowledgement as it travels up the virtual tree of the cluster to be notified.

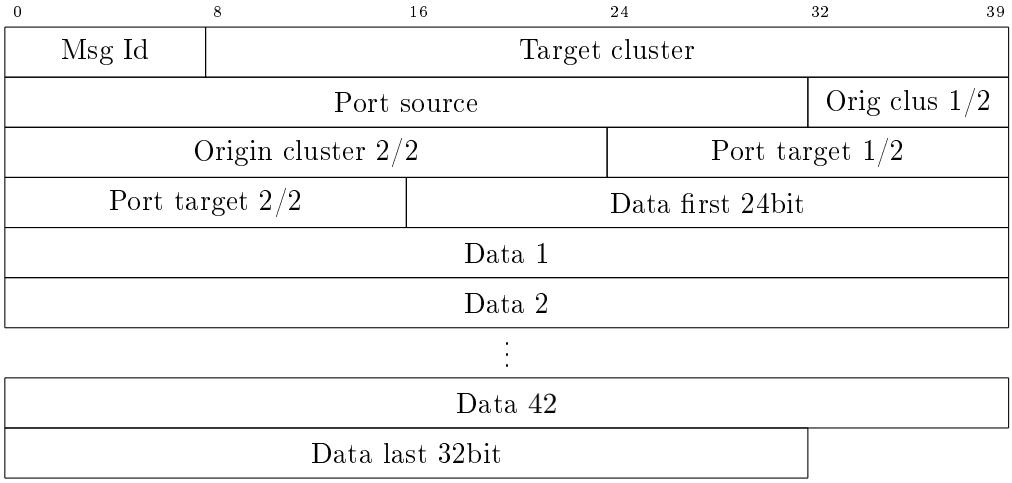
The are a couple of reasons for changing the message id when travelling up the virtual tree of the cluster which is to receive the message. The first one, and most important, is that it allows to diagnose very clearly which amount of packets get lost in the cluster transition by comparing the ratio of the messages with its “2" counterparts. The second reason, is that it allows the code to avoid making certain checks of origin and direction of the packets, a thing that with the limited power of the devices that we target, is to be considered.

The Highway transport messages, depicted in figure 3.3, are also different in length in the two target platforms. However, the difference now cannot be, in this case, just attributed to the different length of the identifiers. Instead, the major difference in this case is the fact that the maximum length of a packet is defined differently by the Radio modules of both iSense and Shawn. Whereas the iSense extended radio defines 116 bytes as the maximum, Shawn's own extended radio allows for 255 bytes. This difference has an impact on the overhead ratio introduced by the highway encapsulation. On iSense the overhead is 9/116 which is just shy of 8% and on Shawn the overhead is 1/15, or just under seven percent.

In figure 3.3a the reader can see the transport packet for the iSense platform. Both in this case and in the Shawn one, depicted in figure 3.3b, the space for the encapsulated message far outweighs the highway protocol headers. This beneficial property, of course, depends on the length of the data that is sent, as the Data field in both pictures is of a variable length determined at sending time by the requirements of the overlying module.



(a) iSense Highway transport packet



(b) Shawn Highway message packet

Figure 3.3: Highway transport packets

The kinds of messages that are sent following this structure are:

- *SEND*: Main highway transport message issued after encapsulating the data.
- *SEND2*: Main highway transport message as it traverses the virtual tree of the target cluster.
- *ACK*: Acknowledgement message issued on reception of a transport message by

the target cluster, the data field is of length zero.

- *ACK2*: Acknowledgement message as it traverse the virtual tree of the source cluster, the data field is of length zero.

4

Implementation

In this subsection, the reader will be presented with a set of diagrams and explanations of the algorithm. Due to the event driven nature of the Highway algorithm, the kind of diagrams chosen are flowchart, which allow for a great case per case visualization, reducing the complexity that a comparable traditional monolithic algorithmic view would show in treating the events.

There are two different implementations of the algorithm, among all the drafts and modifications that were done along the project, that are worth mentioning and describing. The first and original is the one described in section 4.1, and is an implementation based on an extensive use of priority queues inside the highway maps, to allow more complex highway selection. Unfortunately, the space requirements of the iSense platform (when considering the whole network layer stack) demanded saving in both code size and memory footprint, and thus, the one-to-one highway implementation was born.

4.1 Priority Queue based highways

The main characteristic of this implementation is, as the name implies, the use of priority queues. The priority queues, as depicted in figure 4.1, are embedded on the highways maps in a very straightforward way. Per each cluster target, we have a full priority queue (shown in figure 4.2) from which we can get a quite interesting highway picking functionality.

The advantage of using a priority queue for storing the different found paths to another cluster is that we get a cheap load balancing solution and a highway maintenance

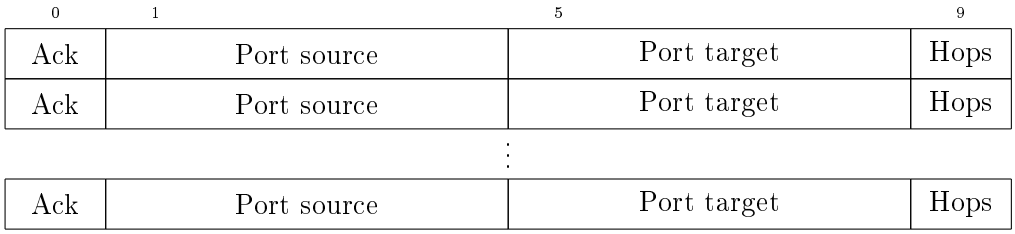


Figure 4.1: Highway priority queue

for a reasonable code size and memory increase[1].

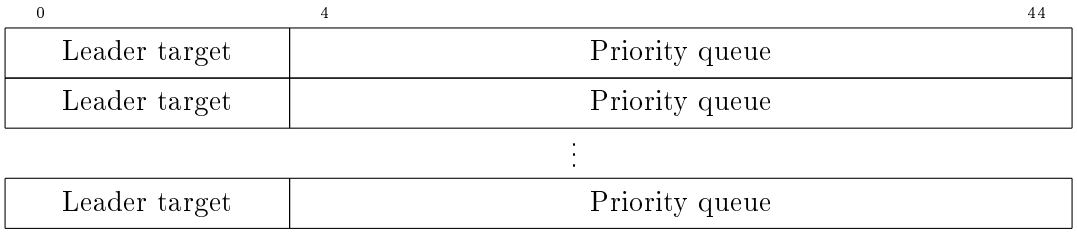


Figure 4.2: Highway map with priority queues

The load balancing is provided, under the circumstance of having equally performing highways established from cluster A to cluster B, by the fact that each sent packet increases the count of acknowledgements that governs the priority queue. Thus, under the assumption that another communication is done from A towards B after the first packet is sent, but before the acknowledgment packet is received, the second communication will be performed through the originally second highway of the priority queue. This partial load balancing, obviously is quite circumstantial, but under high load, or in moments in which a cluster has to deliver a high amount of information, the circumstances can be met.

Under lower loads, the priority queues of the highway maps will just serve as a very convenient list of backups which are ranked according to their previously recorded performance. This characteristic is particularly good in stable topologies in which some nodes go to sleep occasionally due to battery concerns, as after a few lost packets, the first backup highway will get to the top of the priority queue (while keeping the old

[1]The fact that the increase is reasonable for such a functionality does not mean, unfortunately, that it fits hardware when integrating the whole networking layer in the iSense devices.

highway in case it is needed in the future if the node powers back on and is added to the same cluster) and assume the main communication duty.

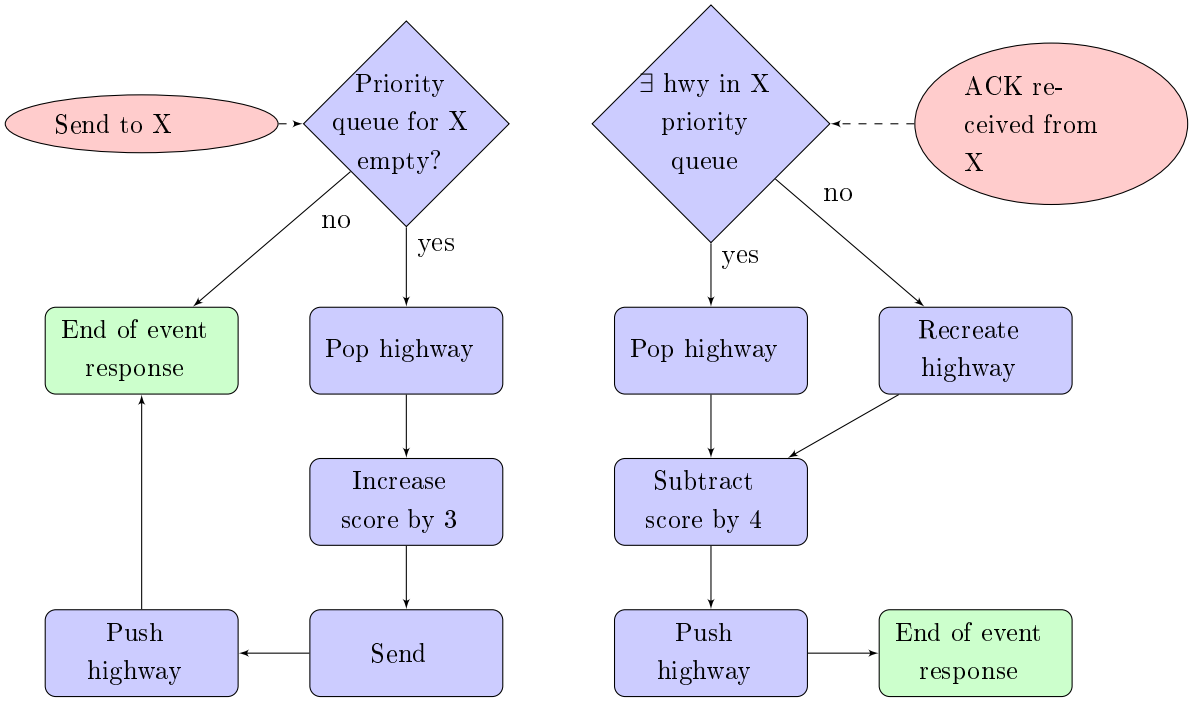
The second benefit from the chosen priority queue implementation in highway maps is that the management of broken highways is automatically performed by our custom displacing push method. The definition of displacing push is a push that, instead of failing to insert an element to the queue when it is full, it performs a comparison with the worst element element of the queue. If the comparison determines the new element to be better, the worst element of the priority queue is dropped, and the new element is inserted in its rightful position (non necessarily the space emptied by the worst element).

During the implementation of the priority queue based highways, however, the goal of the project shifted dramatically putting almost all the importance on the successful porting to iSense along with the rest of the members of the FRONTS networking layer. This change affected the priority queue implementation due to some disadvantages:

- *Priority handling methods*: A part from the displacing push, the implementation required a special pop for retrieving the correct port on PORT_ACK, special handling of the highway scores (a costly pop push).
- *Priority queue data structures*: The data structures used for the highway map with priority queues is about four to eight times bigger than a single highway mapping.
- *Worse performance in volatile topologies*: On starting to test in the WISEBED testbeds, it was observed that when fitting just the highway module and its underlying modules (making some simplifications due to code size constraints) the volatility of the network annihilated most of the priority queue advantages designed for large societies of large and relatively stable clusters[1].

The main differences with the one-to-one implementation found in the next section can be described in a series of flow charts which the reader can see in the following pages. The processes in which the similarity was too big to deserve an increase of the

[1]During this stage of the project and almost until completion, the clustering algorithm did not allow clusters bigger than one hop, jeopardizing most of the sense of utility of the highways, fact that greatly impacted the decision of simplifying the highway implementation as seen in section 4.2.



(a) Priority queue Highway algorithm to send packets through a highway

(b) Priority queue Highway algorithm to update a highway score on acknowledgment reception

Figure 4.3: Priority scores updating.

page count of this document were left on the one-to-one section[1].

Starting the description of the diagrams, in the flowcharts of figure 4.3, the reader can see how the process of sending an encapsulated message through a highway and receiving the corresponding acknowledgement alter the score of the used highway and, potentially, the order of the priority queue. It is important to note that both in figure 4.3a and in figure 4.3b, and due to the Wiselib implementation of the priority queues, we must do a relatively expensive pop and displacing push to update the score. It is costly because the displacing push requires to dump the whole priority queue on the stack as a part of the process as well as performing some reordering depending on the score of the updated highway. On a brighter note, a cluster leader is able to recreate a highway which it had lost if it is able to receive an acknowledgment or highway send packet through it. This recovery is possible thanks to the highway transport message headers, which contain

[1] The rationale behind putting a forward reference like this one is the desire to have a complete walk through the final algorithm implementation, while at the same time being able to explain chronologically the highway implementations this project worked with.

enough information to re-initialize a highway (the old acknowledgement score is lost).

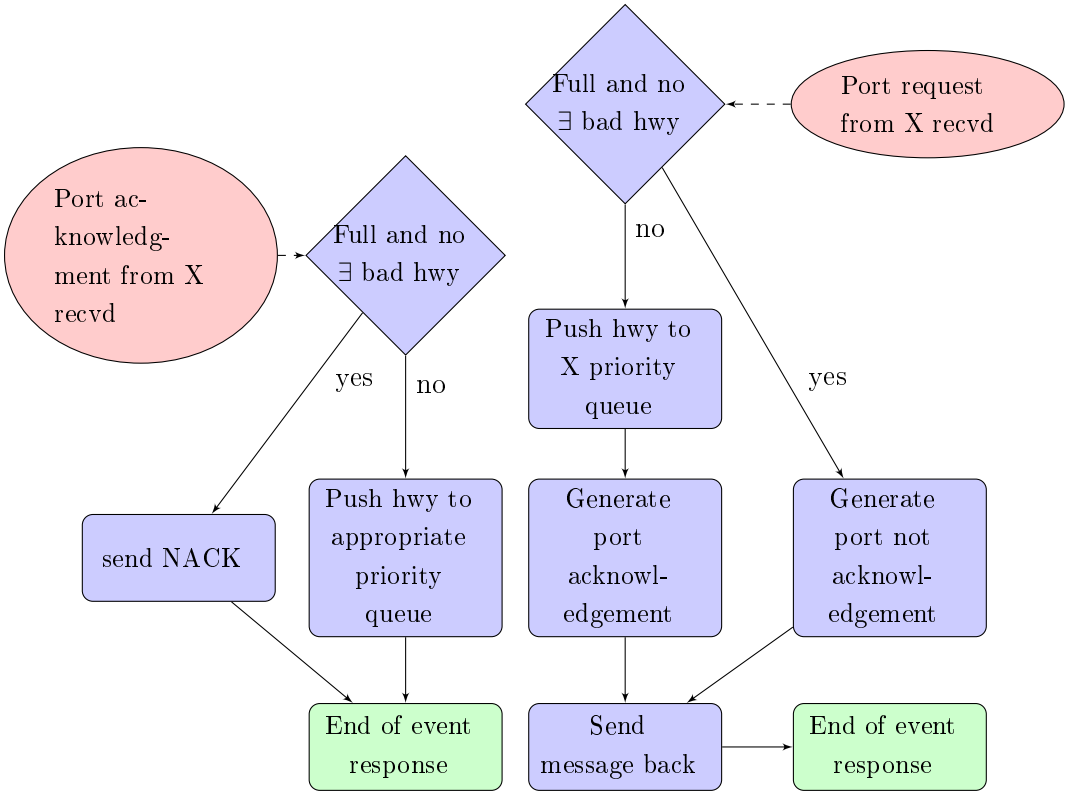
The next batch of diagrams, embedded in figure 4.4, describe the particularities of the priority queue Highway implementation in the process of negotiating highway creation (figures 4.4a and 4.4b) and removal (figure 4.4c).

The port request response in priority queues differs, a part from using a priority queue instead of replacing one field, from the one that is explained in the next section in the fact that the decision for port non-acknowledgement depends on the success of the displacing push. This means that the algorithm will generate a port acknowledgement as long as there is some empty spot in the priority queue or at least one of the spots is held by a highway rated with a below zero performance.

In the case of port acknowledgement handling, the difference is that it can generate an extraordinary non-acknowledgement. It is deemed as extraordinary because normally, a port request is only issued if there is a bad performing highway (score below zero). Under the rare condition in which, on port acknowledgement, return there would be no bad performing highway, the negotiation would be actively broken by issuing a non acknowledgement. This functionality was dropped in later versions in pro of memory reduction, due to the scarcity of the circumstance and the similar results obtained by just pushing the acknowledged highway into the priority queue (which originated the one-to-one solution).

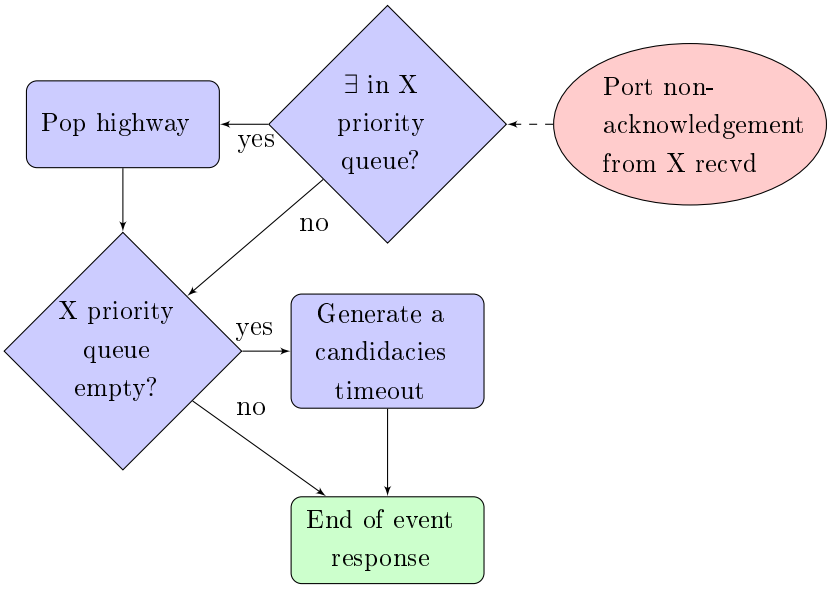
The final diagram, which depicts the case of response to a port non-acknowledgement packet, the main difference is that, because this implementation has a number of backup highways stored in the priority queue, only in the case in which no backup would exist, we would issue an extraordinary candidacies timeout. This solution favours quick stabilization on the first stages of operation, when the priority queue will presumably stay emptier, and avoids unnecessary port requests once the priority queue has been operating with a few highways.

One extra feature of the priority queue implementation respect the one-to-one solution is that the priority queues, being present in all the nodes, allow for failure solving on the ports, i.e., send the transmission through another port of the target cluster if the port specified in the headers is not present in the port priority queue. This could be improved by polling the neighborhood discovery module for port target presence and updating the headers to reflect the change (in case there was a backup highway). This



(a) Priority queue Highway algorithm response to a port acknowledgement

(b) Priority queue Highway algorithm response to a port request



(c) Priority queue Highway algorithm response to a port non-acknowledgement

Figure 4.4: Priority queue Highway algorithm port negotiation.

adds a very interesting degree of distributiveness that, unfortunately, was beyond the memory and code size constraints of the platform.

On a final note, it is important to mention that in the last stages of work in this implementation, where very drastic measures were taken to try to fit the networking layer into the iSense hardware, several modifications were considered and some attempted. They were mainly simplification of the presented diagrams and data structures that eventually were so close to a one-to-one solution, that a refactoring was initiated.

4.2 One-to-one based highways

The one-to-one highway implementation, as explained in the introduction of this chapter, is an implementation that takes some compromises from the higher featured Priority queue based implementation in favor of a leaner code size and an reduced memory footprint. Despite the compromises, this implementation has to comply with the highway design principles stated on section 3.2.

The main memory and code size reduction comes, as the reader can guess, from dropping the priority queues that were embedded in the highway maps, and replacing them with just one entry, as depicted in figure 4.5. The implications of this are varied and, from the point of view of fitting the implementation in the iSense devices, all positives.

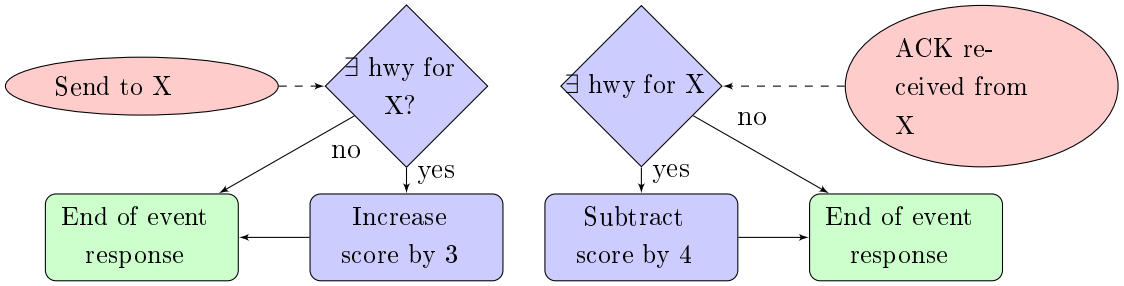
0	2	4	6	7
Leader target	Port source	Port target	Hops	Ack
Leader target	Port source	Port target	Hops	Ack
⋮				
Leader target	Port source	Port target	Hops	Ack

Figure 4.5: Highway map

The first repercussion is shaving all the code of the priority queue class from the compiled firmware. The second is reducing the size of the highway maps roughly eight fold, due to the fact that each target cluster was allowed to have eight highways in the priority queue. It is important to note, though, that during the process of memory

reduction the allowed highways had been shrunk to four entries, making the real memory saving of 4x.

The third reducing factor is shaving the code related to priority queue management, which is easily visible in figure 4.6. This figure shows the updating algorithm for the highway scores that is applied on sending transport packets and receiving their acknowledgements. Comparing diagrams 4.6a and 4.6b with the flowcharts 4.3a and 4.3b, respectively, it is easy to appreciate that there's an important drop in algorithmic complexity (apart from data structures, recreation is dropped to be more conservative on reliability of highways). The dedicated priority queue algorithms that are avoided (without taking into account several controls in the main functions) amount to a couple of completely dedicated methods for performing displacing pushes and pops on the highway maps, as well as simplification of the highway map handling.



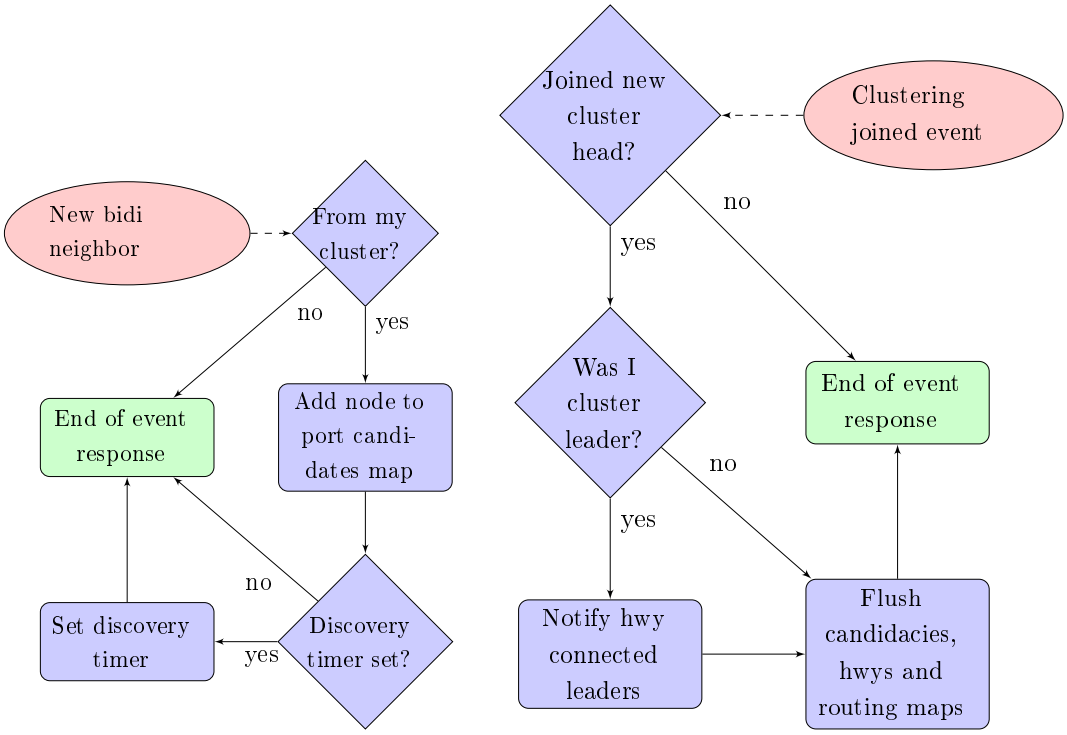
(a) Highway algorithm to send packets through a highway

(b) Highway algorithm to update a highway score on acknowledgment reception

Figure 4.6: Priority scores updating.

Not all are advantages though, and it is expected that some nice features had to be dropped. The most important feature dropped is the one that allowed automatically picking the best working highway, with the side effect of losing also an amount of fall back highways per target cluster that went from three to seven depending on the particular implementation.

On to the details of the implementation, the first diagrams, 4.7a and 4.7b account for the interfaces with the underlying algorithms, the neighborhood discovery and the clustering, respectively. This external events, are successfully received by the Highway Algorithm thanks to the Wiselib's callback registering capabilities implemented by both of those modules. The registering takes place in the enabling routine of the algorithm that, due to it's triviality, will not be diagrammed.



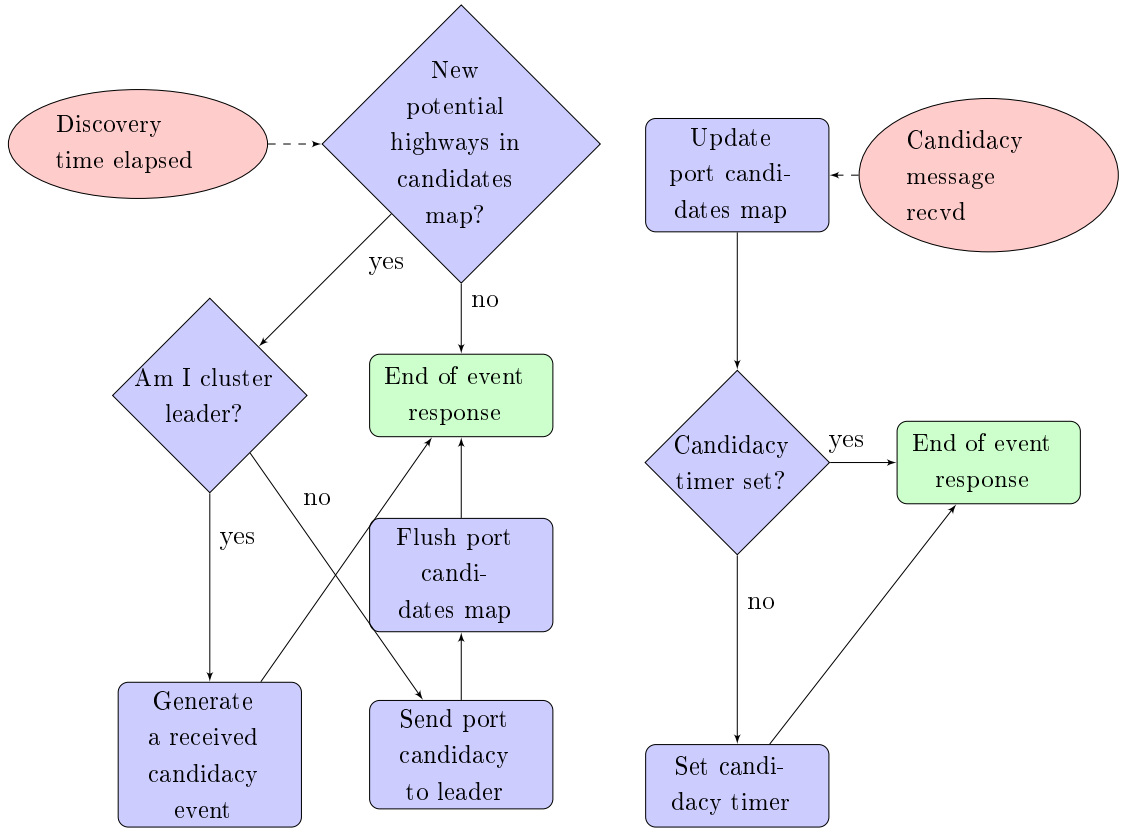
(a) Highway algorithm response to a neighborhood discovery new bidirectional neighbor event

(b) Highway algorithm response to a clustering joined event

Figure 4.7: Algorithm diagram of the response to the clustering and neighborhood discovery related events.

As a result of the previous flowcharts, granted that there are multiple nodes and clusters, the 4.7a decision path will fill the port candidates map and set up a timer to allow for a discovery time window which, on its end, might result in several candidacy requests, as seen in the diagram 4.8a, depending on the amount of changes that happened to the network since the last discovery time window. The candidacy requests, as depicted in the same diagram, will then be sent to the leader or, if the current node is a cluster leader, it will be processed as if the request was sent to itself. Faking the message reception on the leader avoids some extra handling that, otherwise, would be needed to deal with this particularity. The handling of the candidacy messages consists in merely adding them to the candidacies map, and in an homologous way of the 4.7a diagram, sets a message receiving window, as displayed in the 4.8b figure.

Once the candidacies timer elapses its alloted time, it generates an event that is



(a) Highway algorithm response to a discovery timer event

(b) Highway algorithm response to a candidacy request

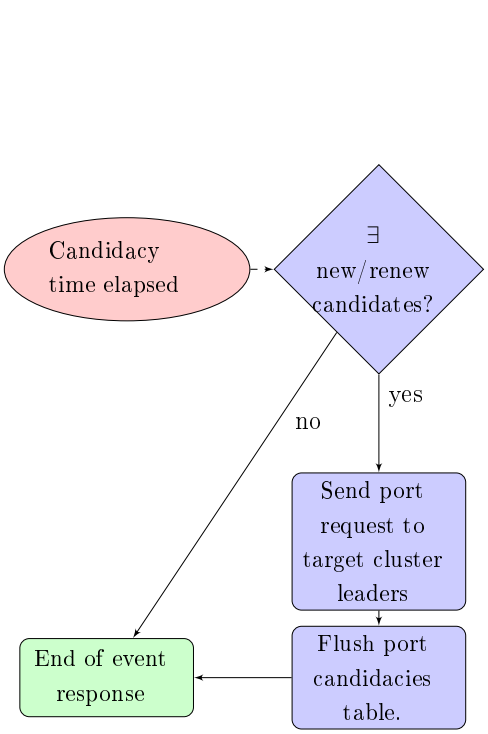
Figure 4.8: Highway algorithm response to discovery and candidacy message events.

processed in the way depicted in figure 4.9a. As the reader can see, it basically checks if any of the highway candidates present real opportunities to create working highways (a renewal of a broken highway or an entirely new highway) and decides to act accordingly, by sending a port requested in cases of map entries with potential. After that, the map is flushed for the next candidacies window to have a clean slate. The port requests are sent to the matching target leader through the highway to be path of nodes and then the algorithm moves on, without blocking nor waiting for any kind of response. The lack of waiting for any kind of response simplifies the solution, allows the devices to operate without any hindrance and, in any case, doesn't bar from treating a response if it is received.

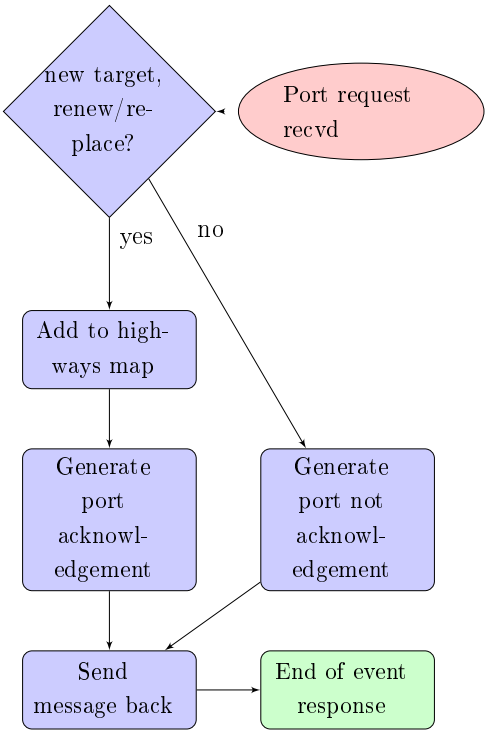
The responses to the port requests can be acknowledgments, non acknowledgements

and no response at all if, due to malfunctions of medium phenomena, the message is lost. The decision process that determines which kind of message will be returned can be observed in the figure 4.9b. The Highway algorithm basically accepts the request unless there is a working highway different to the proposed one. To determine if it is working or not, the algorithm gives and takes points from the highways on the highway map based on successful transmissions. The point give an take is done in an asymmetrical way, taking 3 points per sent message and awarding 4 per received message, this way, long term reliable highways present always a better balance than the clean balance (0).

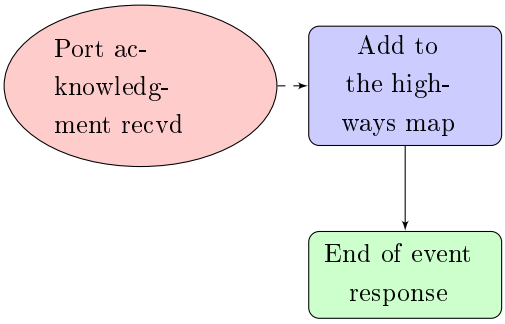
When a node receives an acknowledgment (figure 4.9c), it automatically places the acknowledged highway into its working highways map. In contrast, when a non-acknowledgment is received (figure 4.9d), it is removed from the highways map and the process of the figure 4.9a is invoked without the usual candidacy time window. This is done to try to renegotiate ports on failure, and also because non-acknowledgements are additionally used to signal the end of life of a working highway, i.e., if a leader ceases to function as such, like in the diagram 4.7b, it tries to notify the receiving ends of its now void highways of the invalidity of the paths.



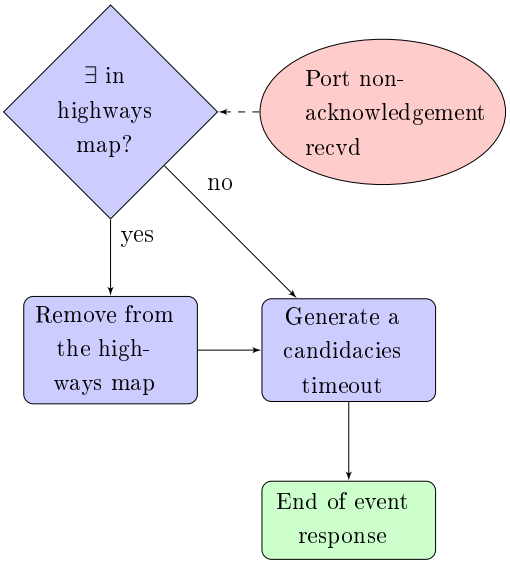
(a) Highway algorithm response to a candidacy timer event on a leader



(b) Highway algorithm response to a port request



(c) Highway algorithm response to a port acknowledgement



(d) Highway algorithm response to a port non-acknowledgement

Figure 4.9: Highway algorithm port negotiation.

5

Experiments

The experimental part of this project had two stages, the developing in fast iterations to bring to reality the design described in chapter 3 and the collection of real hardware metrics for the delivery 4.6[1] of FRONTS and a contribution to a book [23].

The first phase was largely conducted in Shawn with no formal experimentation nor metrics collection. The approach was building prototypes and experimenting different functionalities and data structures. The tail end of this phase was conducted on standalone iSense devices and briefly in WISEBED testbeds.

The second phase was after the debugging and adaptation to the rest of the networking layer. It was conducted along with the development of the tools described in the next chapter. With the aid of these tools and the guidance of Maria Josep Blesa and Jordi Petit, the metrics for the FRONTS project were collected, processed and delivered in the two documents listed in the opening paragraph of this chapter.

In the next sections, the reader will find a brief section about the non formal Shawn simulations experiments and the explanation of the experiments that were held for the FRONTS project.

5.1 Shawn

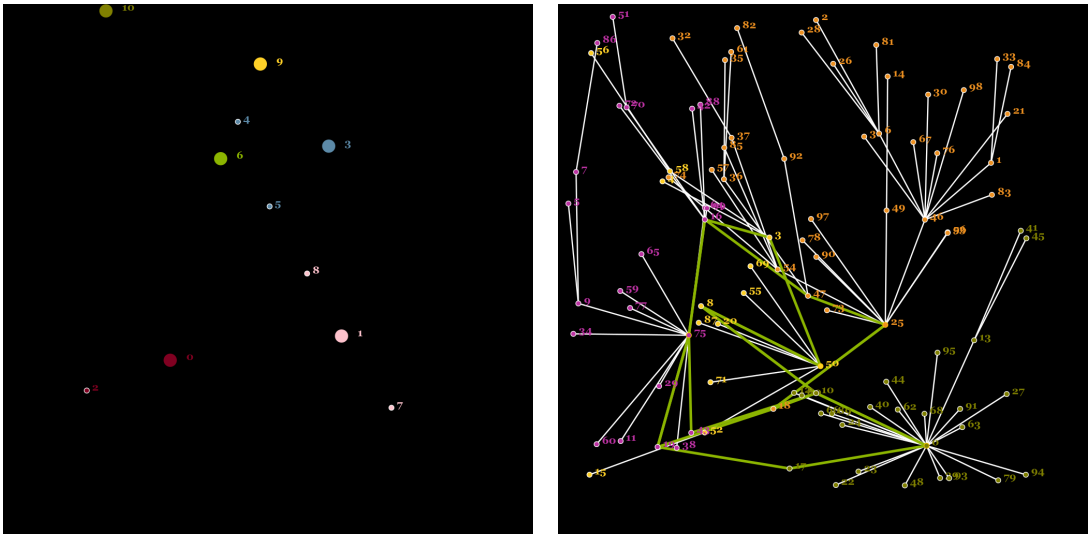
Shawn experimentation, as announced in the introduction of this chapter, was conducted as a way of iterating and refining both the Highway algorithm design and implementation. As such, and due to the fact that the final target, as far as platforms is concerned,

[1] This delivery and others can be found on[22]

was determined to be the WISEBED testbeds, which are not easily modeled in the simulation environment, it was decided to concentrate on using Shawn as a quick and dirty test for functionality and topologies.

The first attempts, as seen in figure 5.1a served only to validate the growing visor tool and to be able to get an idea of the aspect of the random seed topologies for the determined transmission range and model. In the figure, the reader can see how there are six virtual nodes by easily counting the nodes with slightly bigger diameter. These nodes are the cluster leaders from which the highways and the implicit trees should originate. The reason that there are not even implicit trees in the picture is that the graphic corresponds to a stage previous to the determination of the proper places in the code in which to add the special traces for the visor. Because of that, even if the implicit tree was working at this stage, it is not drawn.

On figure 5.1b, the situation is quite different, and both the highways (drawn in green) and the implicit trees (drawn in white) are perfectly visible and show how, at this stage, the algorithm was still using the amount of hops as a main decision maker when negotiation highways. The fact that the clusters in the picture are bigger than two hops, which seems to counter the earlier explanation that the FRONTS clustering didn't implement more than one hop until the late stages of the project, is because 90%

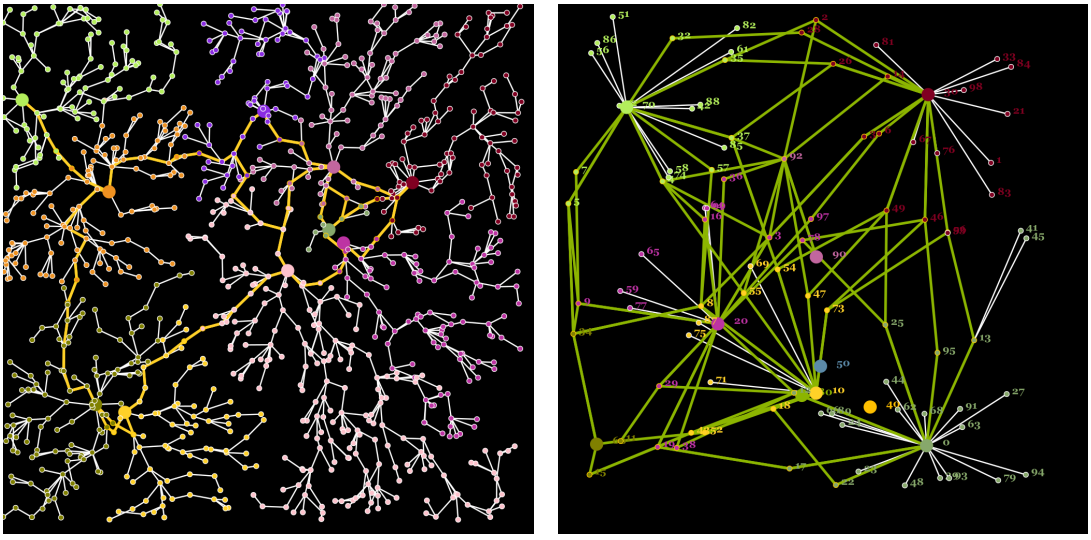


(a) Disconnected topology **(b)** Good connectivity

Figure 5.1: First Shawn experiments

of the Shawn development happened while waiting for the BGU porting[1] from native iSense to Wiselib Shawn.

The choice of clustering algorithm that was made to substitute temporarily the adaptive BGU clustering algorithm was the static bfs routing. Picking a static routing algorithm has two implications, the first one is that it allows for easier testing, as the network stabilizes early on, greatly reducing the use and the edge cases of the network. On the other hand, it doesn't give any hint on the performance of a very important part of the algorithm such as the maintenance one. It is important to note, that the choice was only amongst static clustering algorithms for the simple reason that the Wiselib lacked self-stabilizing algorithms until very late into the FRONTS project.



(a) Large clusters experiment **(b)** Ultra connectivity experiment

Figure 5.2: Other Shawn experiments

With the limitations for generating edge cases imposed by the static BFS clustering, it was decided to try the behavior of the algorithm on networks with substantially bigger clusters with a limited amount of allowed highways, as shown in figure 5.2a and to force the Highway to find a higher amount of paths to use as highways in a small amount of hops (the average is very close to one) per cluster network (figure 5.2b. The first of these two experiments showed clearly the bias of the first implementations of the algorithm

[1]The port did not happen during the life of the project, and FRONTS turned to an alternative developed by CTI, with the inconvenience of a lower degree of theoretic knowledge over it

against longer paths, as between every two clusters, the choice is rarely significantly worse than the minimum path. The second experiment, showed how, even in this adverse conditions, provided the network is stable, the priority queue implementation can build a really favorable network by using almost all the nodes not placed in the extremes for highway communications[1].

5.2 iSense

After the second unifying workshop, held in *La Sapienza*, when the work on integrating the new self-stabilizing clustering and the end to end module began, the experimentation shifted from the simulator to the real hardware, in this case, iSense Devices. At this point, the experiments were conducted by loading the firmware to iSense devices attached by USB to the development machine. The tool used to handle the loading and the posterior experiment trace gathering was iShell.

As the iSense implementation matured and some porting issues were solved[2], the development moved to the WISEBED testbeds. The major part of the initial testbed experiments were conducted on 22 nodes of the *UNIGE* testbed. After the integration of the Layer zero up to the Highway module was achieved with acceptable results, testing took place also in the remaining two testbeds, namely *UZL* and *CTI*. The latter was almost exclusively used only during the third unifying workshop in Patras.

The final stages were completely devoted to collect metrics from more formal experimentation conducted on the three testbeds. The experimentation aimed to collect data for the whole networking layer, but unfortunately, due to device constraints, the end to end module did not fit upon the highway module and was tested separately.

To collect the data the used technology was the WISEBED tarwis software. The experiments were typically run in 5 minutes sessions for the prototyping and tweaking and for 7 to 12 minutes for the experiment trace recording. There is one important point to make on the data gathering, and that is that it had to be run in several sessions,

[1] This could be counterproductive in the case of a highly CPU active higher layer application. This is because the highways and the application would compete for the resources and generally decrease the reliability of both. With lesser highway nodes, the non highway nodes can concentrate more in the data gathering.

[2] Although generally one can recompile Wiselib applications for the different targets without changing the code significantly, some tweaks and adaptations are in order when coming from the simulator (mainly for its tolerance to non standard Wiselib methods).

because the testbed platform has some limitations on the data gathering process. The limitation is that there is a highly direct correlation between the amount of debugging messages (traces) and the probability of the data entry getting lost and not reaching the testbed client. This particularity proved to be quite hard to find out and, when found, to work around[1] to be able to deliver valid metrics for experiments in which three modules were issuing a notable amount of debug messages.

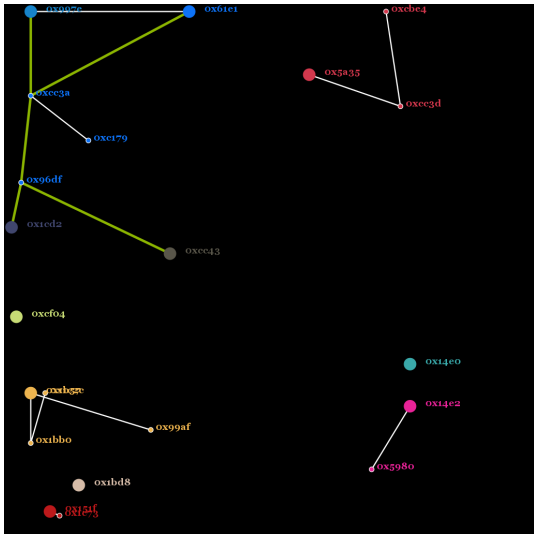
5.2.1 Without node failures

The Highway module sits directly above of the clustering module, and thus runs also on top of the neighborhood discovery module. For this reason, the outcome to be obtained from the experiments is tightly coupled to those from the clustering and neighborhood discovery modules described above. In figure 5.3 we include two snapshots from execution of the Highway module. Depending on the testbed used the resulting topology was connected or periodically disconnected. The higher the connectivity of the testbed, the more important and useful the construction of highways is, not just from a higher chance of delivered packets, but specially when factoring in the increase in cluster stability that results longer living and more stable inter cluster paths. This longevity is important because the shorter the average time for a node to swap cluster or to stand on its own, the bigger the likelihood that a recently formed highway has been interrupted, resulting in cluster leaders trying to deliver data to clusters that are no longer in existence.

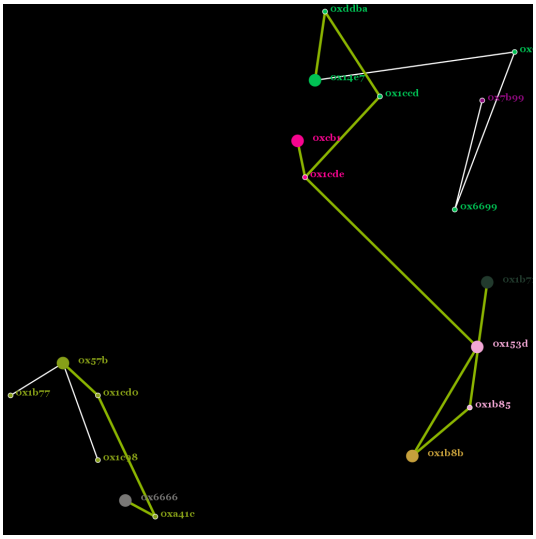
To get an idea how the highways look like on real, and differently distributed networks, we produced some snapshots over three different testbeds, namely, the FRONTS testbed in UZL, the WISEBED testbed in CTI, and the WISEBED testbed in UNIGE. As can be observed from the figures, the connectivity of the UNIGE testbed is higher than in the other ones.

Repeated experiments on the Lübeck FRONTS testbed (see Figure 5.3a)showed very unfavorable results to the Highway module. Those results were due to ever changing clustering configurations while in the process of establishing highways. Their average longevity was found to be short enough so that no high amount of graph connectivity was attained.

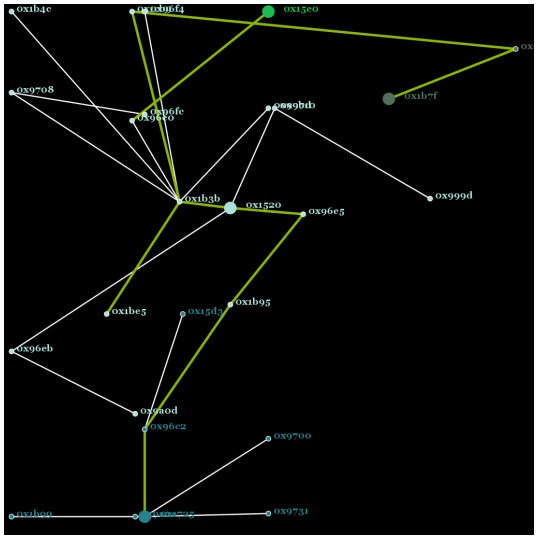
[1] The final workaround was to separate the experiment into the smaller possible units, and inside them test enabling and disabling some traces to be able to gauge the impact of the uncertainty on the experiment.



(a) Disconnected network



(b) Di



(c) Connected network

Figure 5.3: Snapshots of the output of the algorithm for highways' formation on different testbeds. Each color identifies a cluster, where the bigger radius node is the leader. Highways are emphasized in green.

The snapshot of the CTI testbed (see Figure 5.3b), shows clearly how the two main parts of the testbed are fully connected, albeit one node clustering module has not reported yet that its parent has joined the big green top right cluster, and thus believes itself member of a no longer existing cluster. The connectivity between the two halves

of the testbed depicted in the snapshot was rather weak in our experimentation, as links between both parts were rarely reliable enough to establish a two way highway.

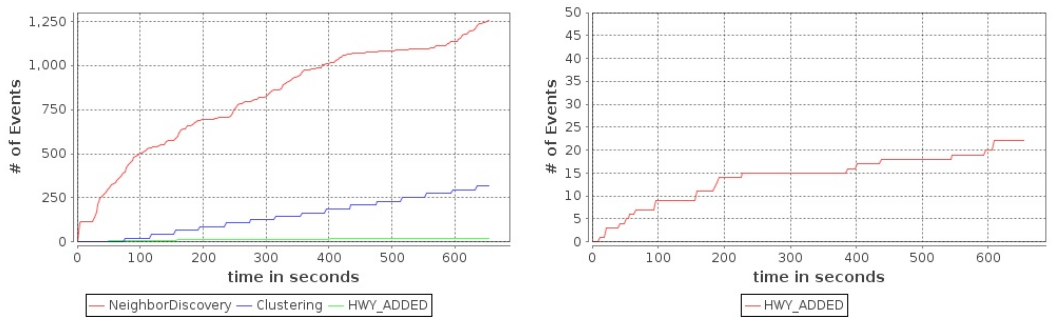
Figure 5.3c shows a snapshot of the UNIGE experiment at an arbitrary point in time after the highways are formed. During the experimentation in that testbed, the degree of connectivity proved itself so big, so that the output messages of highway requests had to be disabled to attain a reasonably small amount of snapshots. Since connectivities are not a static property, the clustering of the network varies also quite often in time, as seen in the previous testbeds results. That means that nodes are continuously re-clustering, and so is their leadership changing. Such dynamic behaviors forces the Highway to recompute new highways connecting the new cluster leaders over the new clustered network, although in this instance, the network proves to be a more beneficial environment for the highways. To see the evolution of this process, the reader is addressed to the following video:

`http://albcom.lsi.upc.edu/fronts/highways-formation.avi`

in which one can appreciate at $5x$ the catch up game between the clustering and the highways.

Concerning now the metrics, we start by studying the amount of events generated by the first three modules of Layer 1. This experiment was configured to track the amount of significant events that each of the modules reported (see Figure 5.4a). The event generated by the Highway module signalled the establishment (or loss) of a highway, i.e., a path between cluster leaders. Note that this message is only generated when the path created has completed a two-step handshake. This means, that if a cluster leader A requests a highway to a cluster leader B, and the latter accepts but the acknowledgement message never makes it back to A, then a temporal one-way path will be established from B to A and no event will be generated.

The algorithm works in a way that the nodes of a cluster A announce themselves to their cluster leader on the occasion that the neighborhood discovery module reports adjacency of extra-cluster nodes. To increase energy savings and medium readiness preservation, an announcement is sent to the leader once for each discovery time. The experiments were conducted with a discovery time of $1000ms$, and during each of these periods the nodes kept track of which were the extra-cluster nodes with a more direct path to each of the neighboring cluster leaders. However, this approach showed itself



(a) Aggregated number of events reported by each module

(b) Detail of the aggregated number of events reporting only highway creation,

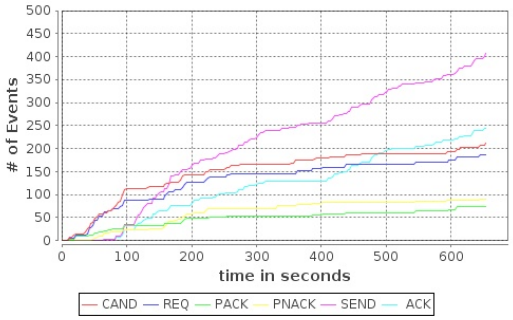
Figure 5.4: Aggregated events experiments ran in the FRONTS testbed.

to be liable to keep *old* candidates and was switched for a newer notice policy, due to the high likelihood of cluster swapping on nodes that lie in certain high traffic points.

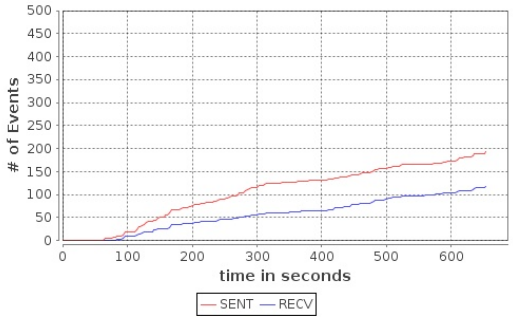
The second experiment tries to factor the energy and medium usage of the Highway module as well as its effectiveness in delivering messages. In this experiment, first we generated highway traffic by waiting $40sec$ initially, to allow the neighborhood discovery and clustering modules to stabilize, and then every cluster leader sent a message to each of the reported highways attached to it. After sending the data, it calculated a random waiting time in the range of $10sec \ldots 30sec$ and repeated the send and wait (with backoff period recalculation). The aim of this is model of traffic is to simulate periodic communication to every neighboring cluster similar to what the end to end module would demand if it were to be run on top of the highways.

Regarding the kinds of Highway messages, it is worth mentioning the hierarchy of messages used within the Highway module (see Figure 5.5a). In the case plotted there, the network was stable enough so that the amount of SEND and ACK messages were well above the highway construction messages. This trend could be reverted by introducing more instability to the medium. After these two kinds of messages, which just propagate around the generated traffic, there is the CAND and REQ messages. The CAND and REQ messages belong to the first steps of the highway construction, respectively informing (their own leader and the other leader they want to connect with) about the possibility of establishing a new highway. In order to reduce the amount of messages (and thus the collisions and message drops), the nodes (specially, the cluster

leaders) filter messages as we go down the hierarchy (e.g., PACK and PNACK account for responses to the REQ messages).



(a) Total number of messages to build highways by type. Nodes that recognize nodes from other clusters periodiacally send CAND messages that are sent to their cluster leader. Leaders process these messages and, if needed, send REQ messages that are propagated to the other clusterhead, which responds with a PACK or a PNACK according to whether it accepts or not the highway. SEND are ACK are the messages that are used to propagate messages though highways between clusterheads.



(b) Aggregated number of messages sent and received through highways.

Figure 5.5: Aggregated messages experiments ran in the FRONTS testbed.

5.2.2 With node failures

We now discuss the effect of node failures on the performance of the Highway module. The experiments were conducted on the FRONTS testbed, in rounds of 15 minutes in which, every five minutes, the running nodes randomly disabled themselves with a 20% chance. As far as traffic and timers are concerned, the same as in the no failures apply, one second of discovery time, and two byte plus highway overhead messages to all neighboring clusters every 10-30 seconds.

As we observe in Figure 5.6, both the clustering and the highways take an obvious hit every time that the amount of nodes is reduced (marked by the vertical bars), although it is appreciated a bigger drop in clusters than in highways, due to a combination of the

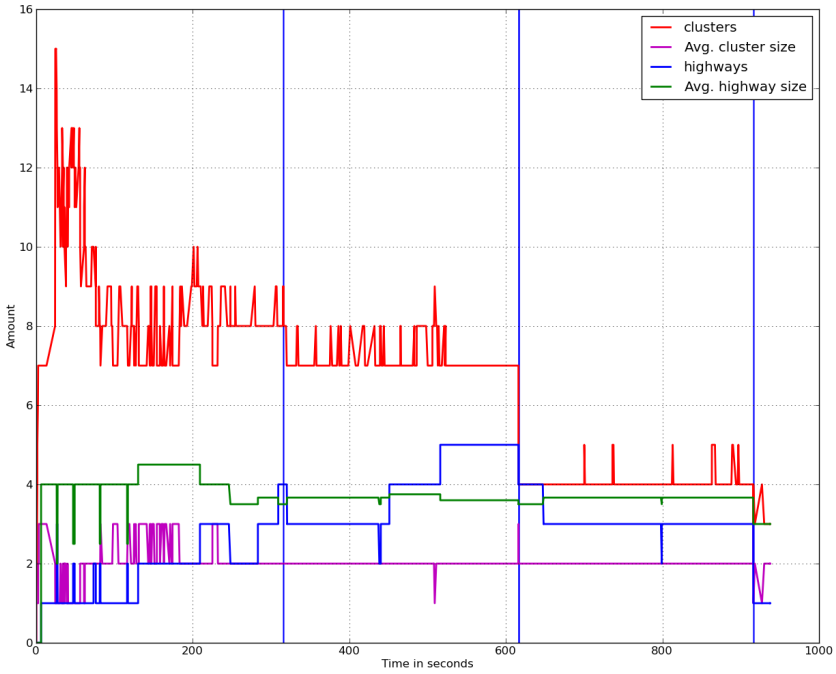


Figure 5.6: Evolution of the cluster amount, cluster average sizes, highway amount and highway average sizes, with the disconnection moments (i.e., 20% nodes failures) marked with vertical bars.

fact that a lost node in the middle of the highway won't be reported until the maximum unreturned ACKs is reached and also due to the fact that in the FRONTS testbed isolated nodes are not uncommon, and these would only take a toll on the clustering number.

Regarding the delivery rate (see Figure 5.7), one can see that in the periods just after disconnections the chasm between sent and received grows, and generates some amount of clustering instability that reduce the delivery rate to about 57% and in general into a 10 to 15 percent less than the fail free version.

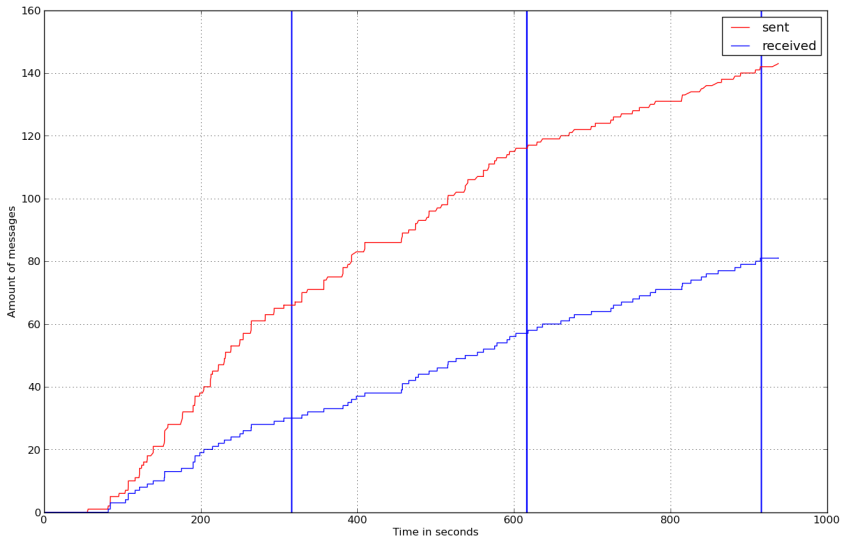


Figure 5.7: Aggregated number of messages sent and received through highways with progressively failing nodes. Disconnection moments (i.e., 20% nodes failures) are marked with vertical bars.

6

Tools and procedures

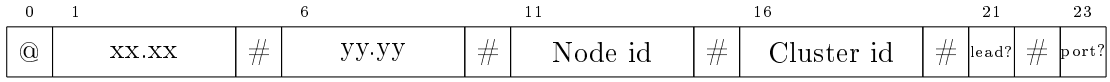
During the experimental stage of the project, either in the simulations or real hardware, the need arose for having a way to generate snapshots of the topology and module status. Initially, the vis module of the FRONTS' simulator of choice, i.e., Shawn, was considered. However, its adoption needed a considerable amount of customization effort, which would be of no use once the project would move on to real hardware experimentation that, as seen throughout the document, is the real aim of the project.

To address the snapshot generation issue, the FRONTS management proposed Spyglass, a Java visualization software developed by the Universität zu Lübeck. However, it was deemed not ready to meet the requirements nor to adjust to the purpose and the integration process was dropped. As a last solution, CTI produced an ad-hoc Java OpenGL visor which worked on traces to produce an animated display. To display the highways, though, the "vizor" did not produce optimal results as far as highway visualization is concerned and so, we decided to perfect the implementation of a visualization tool in Python which had started being developed during the first unifying workshop in Braunschweig.

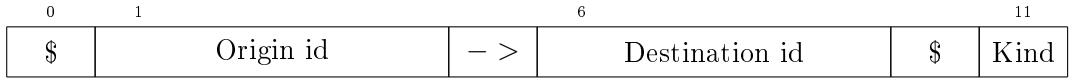
6.1 Static visor

The static visor is designed to work with the output of the Shawn simulator. The functionality is simple yet powerful. It consists on parsing a special format of debug messages which must be issued upon certain events to provide enough information to the visor engine to draw the topology, highways and implicit trees.

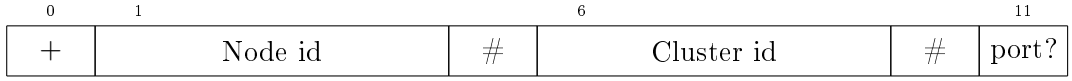
There are several revisions of the special message format, but the latest implementation is as follows:



(a) Node initialization message



(b) Edge message



(c) Update message

Figure 6.1: Static visor special messages.

This messages are deployed in the initialization method (figure 6.1a), reception of a port acknowledgement (figure 6.1b) and on the transmission of a port acknowledgement and candidacy (figure 6.1c). In the three cases, all the symbols are needed separators that ease the parsing task. The fields with a question mark should be filled by a zero or a one, not by boolean values. Finally, the kind field should contain a t when the edge is part of an implicit tree and h when it is a highway edge. Maybe the reader will have noticed that there is no syntax for removing highways, and she or he would be right. Since it was developed for static clustering under a simulated environment, the possibility of dropping a highway is almost inexistent.

This tool is built on Python and uses the Cairo bindings for creating a Portable Networks Graphic or a Scalable Vector Graphic file. Other Python modules that it uses is the *re*, for the regular expression matching; *optparse*, for allowing the user to make some choices regarding the output file; *math*, for the viewport transformations; *sys*, for getting the arguments from the system and *random* for the color picking..

The visor supports up to sixteen different clusters, although adding more color setting methods of changing the color picking by a total random function (as used by the dynamic visor) could shed the cluster limit, leaving the software only bound by the limits of the Python interpreter and the hardware it is running on. The semi manual

choice of color answers the requirement of getting always very differentiated cluster colors.

There is only one accepted parameter for the visor, and that is the output file from executing a Shawn experiment with the appropriate traces enabled. In regards to options, those that this application makes available to the user can be checked by calling the visor with the *-h* parameter. In the moment of writing they are:

- *-h* | *–help*: Displays these options.
- *-s* | *–show_ id*: Draws the node id next to the node circle.
- *-r* | *–random_ colors*: Shuffles the default cluster color assignment order.
- *-b* | *–bigger_ leaders*: Draws significantly bigger node circles for the cluster leaders.
- *-y* | *–yellow_ highway*: Swaps the default green color with a strong yellow for drawing the highways.

This application is free software and is made available under the GPLv3 license. To check it out, one can clone its git repository issuing the following command:

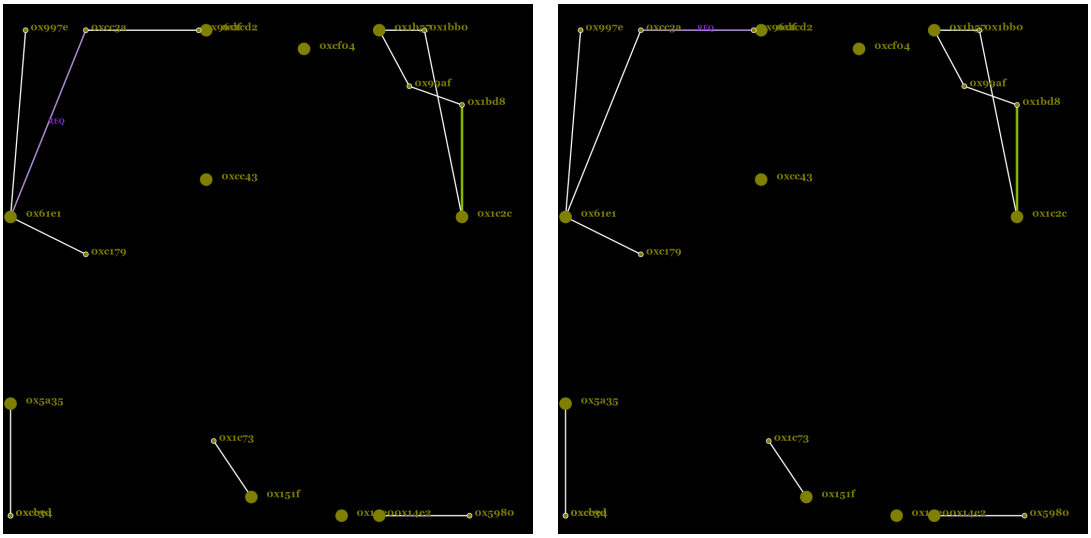
Listing 6.1: Cloning the static visor repository

```
1 git clone https://github.com/celebdor/WSN-visor.git
2 cd WSN-visor
3 git checkout master
```

6.2 Dynamic visor

The dynamic visualization tool, that goes under the name of WSN-visor, makes extensive use and is highly integrated with the Python graph analysis software called NetworkX. This integration enables for quick and easy extensibility in adding to the visor graph property evaluation features such as diameter connectivity, etc. Another useful advantage of using NetworkX is that it contains several position generations that are very useful when visualizing a testbed for which some topological information is missing or has changed.

As the static visor, WSN-visor uses the Cairo library to render a snapshot of the wireless sensor network. In this case, however, the algorithm does not consolidate a bunch of special messages but reacts on every significant highway event to modify the



(a) Dynamic visor sample graph 1 **(b)** Dynamic visor sample graph 2

Figure 6.2: Snapshots of the output of the algorithm for highways' formation on different testbeds. Each color identifies a cluster, where the bigger radius node is the leader. Highways are emphasized in green.

NetworkX multi graph object and renders a new snapshot. The monitored events are not just the ones generated by the Highway module but also those coming from the clustering module.

The snapshots, as explained above, have a one to one relationship with the event traces that come from the testbed. This allows the algorithm to assign every snapshot a timestamp and a sequential[1] number. On the end of the process, from all the snapshots (like the ones depicted in figure 6.2), which are saved into a results folder, the user can use a small BASH tool to make an accelerated movie of the network evolution using mplayer's mencoder tool. Thanks to these movies, one can improve the understanding of the experiments by seeing how the different modules react and cooperate in front of changes in connectivity.

WSN-visor, like its static counterpart, also allows for some customizations, like passing the tool comma separated values files with the node positioning, choosing how the coloring is performed, the sizes of the leaders, displaying or not the node id's, etc. Additionally, the snapshots can be chosen to be vector drawings, which make ideal

[1]The testbed software cannot guarantee the strict order of the relayed messages, and thus, it is common to see logical glitches in the generated movies as a result of wrong event order

targets for large network visualization, by enabling limitless zooming and panning. The command line options are:

- *-h* | *–help*: Displays these options.
- *-s* | *–show_id*: Draws the node id next to the node circle.
- *-l* | *–show_label*: Draws the message edges with their message identifier.
- *-r* | *–random_colors*: Shuffles the default cluster color assignment order.
- *-n* | *–random_position*: Uses NetworkX to generate the node layout.
- *-b* | *–bigger_leaders*: Draws significantly bigger node circles for the cluster leaders.
- *-y* | *–yellow_highway*: Swaps the default green color with a strong yellow for drawing the highways.
- *-p* | *–png*: Generates the snapshots in Portable Network Graphics (needed to make movies).
- *-f PROP* | *–prop_file PROP*: Get the node positions from the PROP csv file.
- *-o OUTFILE* | *–outputP OUTFILE*: Specify the name of the base output file(recommended inside a new folder).
- *-v* | *–verbose*: Dumps extra info on the standard output.

To check out the software and generate a movie after processing an experiment, it can be done as follows:

Listing 6.2: Generating a movie from the snapshots

```
1 #If it has been cloned for the static version, skip to line
     5.
2 git clone https://github.com/celebdor/WSN-visor.git
3 cd WSN-visor
4 #Regardless of the name, the branch is in beta stage
5 git checkout experimental
6 #foldername is the name of the folder which was passed to
     the visor for dumping all the snapshots
7 ./graph2movie foldername
```

6.2.1 Metrics generator

To generate the figures, a similar solution (as far as performing non-gui trace parsing in python) to the visor tools was developed and included in the WSN-visor distribution.

In this utility application for generating graphs of the metrics, though, Cairo was not directly used, and instead the drawing functionality was implemented using the semi interactive pyplot module of the Matplotlib. The source code of the application can be found on the *graph_generator* directory.

The grapher, as it was non-originally labeled, takes the trace file as only input, and allows the user to generate three different kind of graphs relevant to the analysis of the highways such as delivery rate, delivery rate percentage and clustering versus highway evolution. Due to the testbed software message collection uncertainty principle explained earlier, one can rarely perform all the graphics from just one experiment, and often the disjunctive is between exhaustive highway metrics or reduced highway metrics in comparison to the other modules. The command line options for picking the kind of graph have the following syntax:

- *-h* | *–help*: Displays these options.
- *-r* | *–delivery_rate*: Makes metrics of sent and received instead of sizes.
- *-e* | *–percentages*: Shows the delivery rate in percentage.
- *-p* | *–plot*: Plots the data using matplotlib.

The last of the command line switches, allows for a text based analysis, for getting a quick outlook on the data. This functionality was specially useful during the development stages in which, due to the testbed software needs of a public open listening port, the work had to be conducted through a terminal only secure shell session.

A key advantage of the grapher is the ability to get a GUI window with the graph for review, that allows for modification of the output. The more regular modification is to zoom into a specific region of the graphic and save both the general and the specific images. This functionality is completely provided by the Matplotlib leaving the parsing and experiment logic as the core functionality to be developed.

7

Economics

The aim of this chapter is to give the reader an accurate assessment of the planning and economic considerations that were necessary for the completion of the current project. Thus, it has been split into two sections, namely, planning and project costs.

7.1 Planning

The planning for the project of designing and implementing Highway construction is tightly coupled to the planning of the experimental part of the FRONTS project. As a consequence, the planning presented in the pages 91 and 92, is highly dependent on the FRONTS milestones such as the unifying workshops and the deliveries/publications.

The chosen format for the planning presentation is the Gantt diagram, as it allows the reader to get a clear idea of the time frame of the different aspects of the project with little more than a brief look. As a disclaimer, it is important to note that the implicit calendar that this Gantt represents is not the one derived from the initial planning of the FRONTS project, but the calendar in which the events finally happened. This is an important fact, because due to the pioneering properties of the project and the involved technologies, there were a number of setbacks that forced the coordinator to request a three months extension to achieve a proper delivery.

The planning has been structured around six main axis, that correspond to the several working fronts of the project:

- Algorithm design: The process of getting familiar with the problem and designing a general algorithm to solve it.

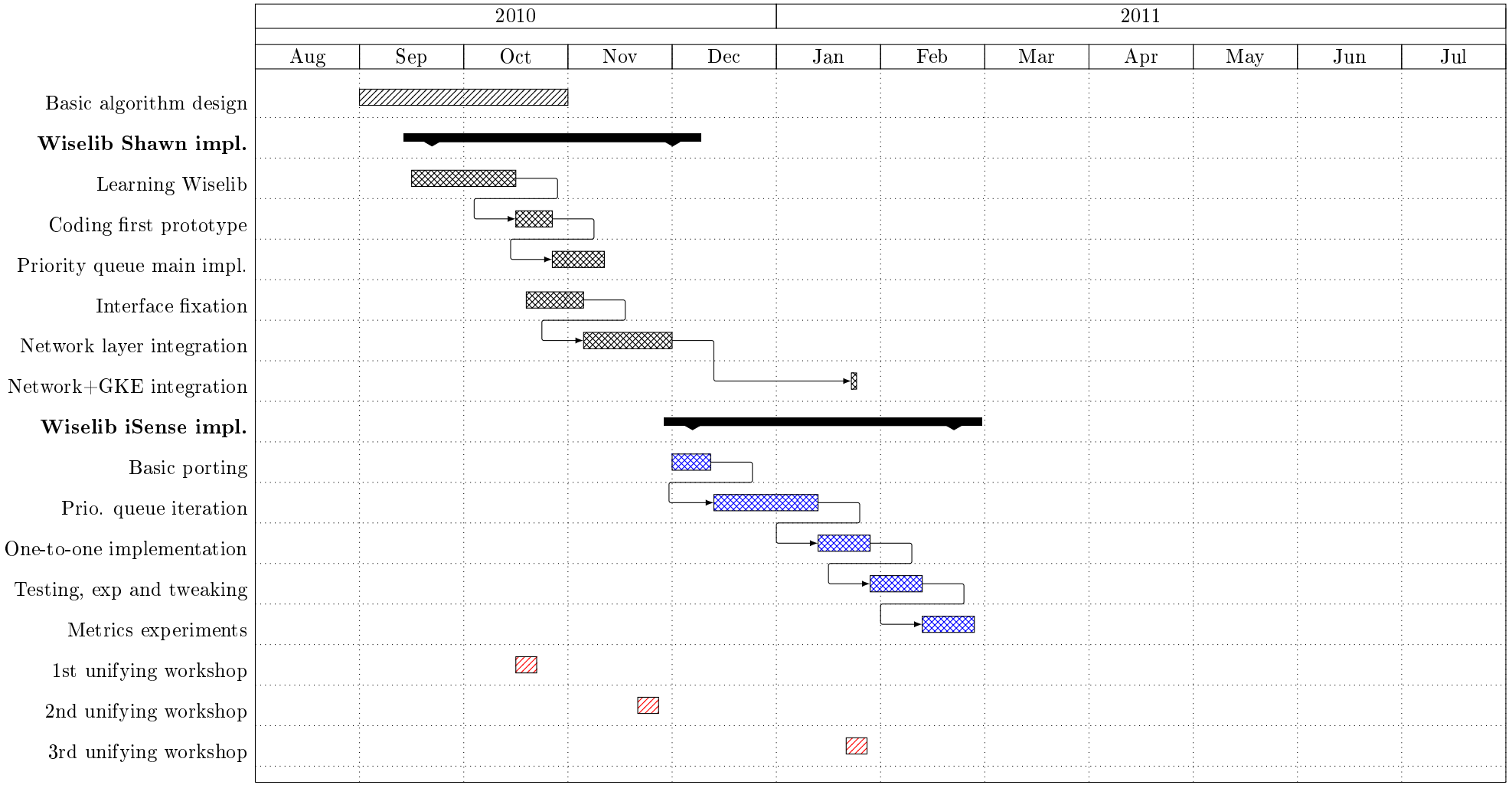
- Wiselib Shawn implementation: All the steps involved in producing a working implementation of the Highway module with the Shawn simulator as a target.
- Wiselib iSense implementation: All the steps involved in producing a working implementation of the Highway module with iSense and the WISEBED testbeds as a target.
- Tool generation: The developing of tools to aid in the research and administration of the development environment.
- Documentation: The documenting effort on FRONTS, WISEBED and Barcelona Tech Final thesis.
- Workshops: The participation in the FRONTS unifying workshops.

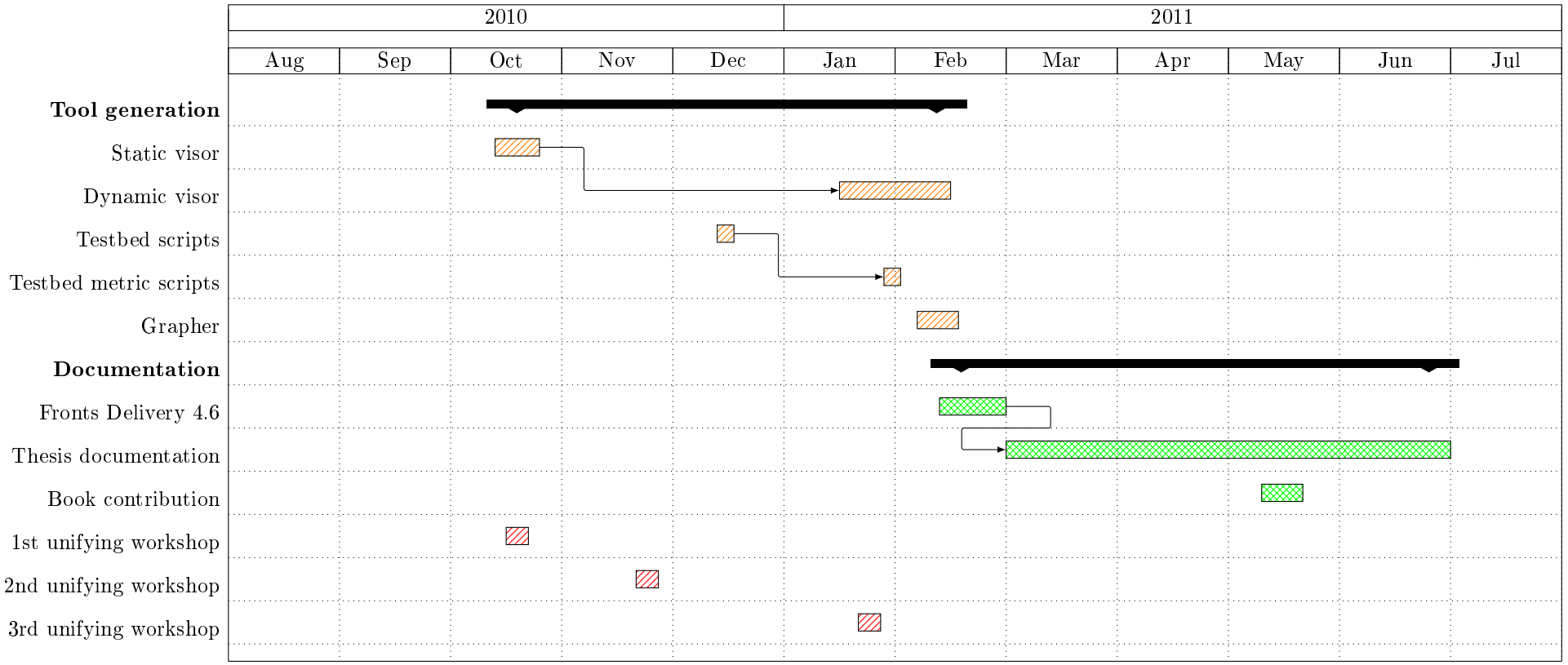
The first Gantt is devoted to the design and implementation steps, which account for the first three bullet points of the previous enumeration; while the second Gantt shows the tool generation and documenting efforts. The unifying elements which help putting in perspective all of these processes are the workshops and, for this reason, they have been included as essential references to both Gantt diagrams.

The information to extract from the first Gantt would be the importance of the unifying workshops as milestones and turning points for the project, as in each occasion they meant taking the project to a new level. On the first workshop, held in Braunschweig, facilitating the prototyping and setting the bases for interface and concepts integration; on the second one, held in Rome, providing the final steps into the simulated Shawn Layer zero integration (adding the backup self stabilizing clustering) and setting the foundations for the iSense porting; and finally, on the workshop held in Patras, the redesign of the Highway module into a smaller solution to meet the platform constraints as well as the FRONTS integration requirements. In the last workshop a rather important turning point was the start of intensive[1] experimentation on the testbeds.

The second Gantt shows a similar picture, with the tools generation happening basically because of the workshops and the deliveries. If one takes both Gantts side by side, it is easy to see that the documenting effort started shortly after the 3rd unifying workshop, as that meeting helped greatly into shaping up the final conditions and requirements for the experiments from which the delivery and the book were produced.

[1]Experiments on testbeds had been previously conducted, albeit with mainly debugging and redesigning purposes


2010
2011
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Jun
Jul
Basic algorithm design
Wiselib Shawn impl.
Learning Wiselib
Coding first prototype
Priority queue main impl.
Interface fixation
Network layer integration
Network+GKE integration
Wiselib iSense impl.
Basic porting
Prio. queue iteration
One-to-one implementation
Testing, exp and tweaking
Metrics experiments
1st unifying workshop
2nd unifying workshop
3rd unifying workshop


2010
2011
Aug
Sep
Oct
Nov
Dec
Jan
Feb
Mar
Apr
May
Jun
Jul
Tool generation
Static visor
Dynamic visor
Testbed scripts
Testbed metric scripts
Grapher
Documentation
Fronts Delivery 4.6
Thesis documentation
Book contribution
1st unifying workshop
2nd unifying workshop
3rd unifying workshop

7.2 Project costs

In this section, the reader will find out an estimation of the costs incurred in the progress of the project. These costs, shown in table 7.1 are the approximate amounts that were paid and thus, do not account for the extra time involved in the project nor the documentation efforts after March 31. To make an assessment of what the extra costs would have been added should the extra time and the documenting be considered, it is fair to assume about a fifty percent increase on three months of scholarship and two months more of work, that could be accounted at scholarship price. Thus, the extra capital required to account for these hidden costs would be of 4550€.

Item	Unit cost	Amount	Total cost
Frühlings hotel	88	6	528
Plane to/from Braunschweig	358	1	358
Train Dresden Braunschweig	42	1	42
Train Braunschweig Dresden	62	1	62
Expenses Braunschweig	130.53	1	130.53
Rome plane	165	1	165
Rome hotel	490	1	490
Rome expenses	180	1	180
Patras workshop	546	1	546
Patras hotel	54	5	270
Greece car rental	350	1	350
Monthly scholarship	700	6	4200
iSense devices	225	6	1350
Total			8671.53

Table 7.1: Table showing the costs of the this project

One final note in this chapter, the costs in table 7.1 do not reflect the amounts spent in the expenses of other participants of Barcelona Tech which were also contributing to the project. These participations cost could be roughly estimated as doubling all the workshop attendance costs to account for Victor Lopez Ferrando collaboration. Also adding 50% to the Rome workshop attendance for Jordi Petit Silvestre and adding 100% to the Patras workshop costs minus the rental for the collaboration of Jordi Petit Silvestre and Maria Josep Blesa Aguilera.

8

Conclusion

As explained throughout this thesis, the project of designing and implementing Highways has been developed in the framework of European Union field pioneering projects. This characteristic is a sharp two edged sword that on the first side allows for great freedom to come up with novel algorithms and techniques to address problems that lay in the mid term future, while the second side places constantly evolving and iterating technology to emulate this future reality, resulting in facing many undocumented challenges, uncertainty and other setbacks. This, of course make the current project into something more challenging, but at the same time much more instructing and gratifying than the regular same old day to day refurbish technology Engineering thesis.

Reviewing the technology used throughout the project, and with the distance of more than a month since having to operate with it on a daily basis, there are some lessons to be learnt but a lot of reasons to feel confident about wireless sensor networks.

Starting with the lessons that should be taken, and as usual in groups lacking technical writers, there should be a documenting effort for the core technologies of the field several orders of magnitude bigger than there is today. The second point is tightly related to first and it is the need for increasing inter site collaboration either through an increase in workshops (the schedules of lecturers make this hard to achieve) or by adopting online technologies for sharing project knowledge, goals, bugs, findings, etc., as well as setting up weekly code reviews between interacting modules to encourage collaboration and understanding. Even if this higher collaboration would translate to a weekly 10% of what was achieved in a week in any of the workshops, it would mean a raise of the research potential to a whole new level.

On a more technological aspect, the lessons that should be taken are to conduct more early stage prototype experiments to get a better feel for the real characteristics and pitfalls of the target platform, and to attempt to assess a priori the real capabilities of the target hardware such as its resources versus the amount of software that has to be embedded in it.

In regards to the reasons to have a positive expectation regarding the future of wireless sensor networks, one must mention the achievement of several interesting proofs of concept, in the form of FRONTS modules. Also the mostly successful experience with the testbed software, which proved that with a little bit better communication and feedback, it is to become a very valid experimentation model. The final positive aspect I would like to note is the tremendous potential that this computing model can achieve with a reasonably small, compared to the global trend of improvement of performance per watt and production cost, increase in the hardware processing and communication capabilities.

On a more personal note, the conclusions I extract from the project presented in this document are basically the good potential shown by the Highway module with stable clustering, the reasonable, but much less than ideal performance on unstable topologies, and the lesson of pushing more for collaboration and early disclosure of the performance status and expectations of collaborating modules and hardware (both iSense as standalone devices and the testbeds as an agglomeration of sensors).

Summarizing, the conclusion of the project is that there is a lot of research to be made in terms of dealing better with highways response instability of the networks, large societies of real devices, long standing experiments and power efficiency experiments; and that the work done and shown in this document proves that it is justified to dedicate resources and efforts to pursue these next level opportunities for research.

Appendix A

Application source code

Listing A.1: Source code of one-to-one highways application.

```
1  /*
2
3   * Test different clustering algorithms
4   */
5  /* Stabilizing cluster includes */
6  #include "external_interface/external_interface_testing.h"
7  #include "algorithms/cluster/fronts/fronts_core.h"
8  #include "algorithms/cluster/modules/chd/attr_chd.h"
9  #include "algorithms/cluster/modules/it/fronts_it.h"
10 #include "algorithms/cluster/modules/jd/bfs_jd.h"
11 /* End of stabilizing cluster includes */
12 #include "algorithms/neighbor_discovery/echo.h"
13 #include "algorithms/cluster/highway/highway_oh.h"
14 #include "internal_interface/routing_table/
       routing_table_static_array.h"
15 #include "../pltt/PLTT_UNIGE_topology.h"
16 //#define HWY_APP_DEBUG
17 //#define GENEVA_TESTBED
18 #ifndef CTI_VISOR
19 #define CTI_VISOR
20 #endif
21
22 typedef wiselib::OSMODEL Os;
```

```
23 typedef Os::TxRadio Radio;
24 // ----------------------------------------------------------
25 typedef wiselib::Echo<Os, Radio, Os::Timer, Os::Debug>
      neighbor_t;
26 typedef wiselib::StaticArrayRoutingTable<Os, Radio, 15>
      RoutingTable;
27 /* Stabilizing cluster Type definition */
28 typedef wiselib::AtributeClusterHeadDecision<Os, Radio>
      CHD_t;
29 typedef wiselib::BfsJoinDecision<Os, Radio> JD_t;
30 typedef wiselib::FrontsIterator<Os, Radio> IT_t;
31 typedef wiselib::FrontsCore<Os, Radio, CHD_t, JD_t, IT_t>
      clustering_algo_t;
32
33 typedef Os::Timer::millis_t millis_t;
34 typedef wiselib::HighwayCluster<Os, RoutingTable,
      clustering_algo_t, neighbor_t, 8> highway_t;
35 /* End of stabilizing cluster Type definition */
36
37 class ClusteringApplication {
38 public:
39 #ifdef VISOR_DEBUG
40 typedef Os::Position::position_t tipus_posicio;
41 #endif
42      void init( Os::AppMainParameter& value )
43      {
44           // Get the modules to make the highway work
45           radio_ = &wiselib::FacetProvider<Os, Radio>::
                get_facet( value );
46           timer_ = &wiselib::FacetProvider<Os, Os::Timer>::
                get_facet( value );
47           clock_ = &wiselib::FacetProvider<Os, Os::Clock>::
                get_facet( value );
48           debug_ = &wiselib::FacetProvider<Os, Os::Debug>::
                get_facet( value );
49           rand_ = &wiselib::FacetProvider<Os, Os::Rand>::
```

```
            get_facet(value);
#ifdef VISOR_DEBUG
            position_ = &wiselib::FacetProvider<Os, Os::
               Position>::get_facet( value );
#endif
            // Initialize values for the highway construction
            highway_.init( *radio_, *timer_, *clock_, *debug_,
                *rand_, clustering_algo_, neighbor);
            highway_.set_discovery_time(700);
            highway_.enable();

            // These two lines enable the debug output of
               their respective modules.
            //neighbor.register_debug_callback( 0 );
            //clustering_algo_.register_debug_callback();

            // Register the callback for highway receiveing
            highway_.hwy_reg_recv_callback<
               ClusteringApplication, &ClusteringApplication::
               receive>( this );

            // Set the highway test application to start after
                the clustering has been done.
#ifdef VISOR_DEBUG
#ifndef GENEVA_TESTBED
            tipus_posicio pos = position_->position();
#else
            tipus_posicio pos = <tipus_posicio, Radio>
               get_node_info( radio_ );
#endif
#endif

#ifdef VISOR_DEBUG
            debug_->debug("@%f#%f#%d#%d#%d#0\n", pos.x(), pos.
               y(), radio_->id(), clustering_algo_.cluster_id
               (), clustering_algo_.is_cluster_head());
```

```
76 #endif
77            timer_->set_timer<ClusteringApplication, &
                 ClusteringApplication::start>( 40000, this, 0 )
                 ;
78       }
79       // -------------------------------------------------------
80       void start( void* )
81       {
82        if( clustering_algo_.is_cluster_head() )
83            {
84                 typedef highway_t::Node_vect VN;
85                 typedef VN::iterator vn_it;
86
87                 highway_t::Node_vect neigh;
88                 highway_.cluster_neighbors( &neigh );
89                 vn_it it;
90                 for( it = neigh.begin(); it != neigh.end();
                      ++it )
91                 {
92 #ifdef CTI_VISOR
93                      debug_->debug("CLUSTERING_OH; SENT; %x;
                         %x", radio_->id(), *it );
94 #endif
95                      uint8_t buffer[2];
96                      buffer[0] = 42;
97                      buffer[1] = 14;
98                      highway_.send(*it, (size_t)2, buffer);
99                 }
100           }
101           int backoff = (*rand_)(20000);
102           backoff += 10000;
103           timer_->set_timer<ClusteringApplication, &
                 ClusteringApplication::start>( (millis_t)
                 backoff, this, 0 );
104      }
105
```

```
106       void receive( Os::Radio::node_id_t from, Os::Radio::
              size_t len, Os::Radio::block_data_t *data )
107       {
108  #ifdef CTI_VISOR
109                       debug_->debug("CLUSTERING_OH; RECV; %x;
                             %x", radio_->id(), from );
110  #endif
111       }
112
113  private:
114       neighbor_t neighbor;
115       highway_t highway_;
116       Radio *radio_;
117       Os::Timer *timer_;
118       Os::Clock *clock_;
119       Os::Debug *debug_;
120       Os::Rand::self_pointer_t rand_;
121  #ifdef VISOR_DEBUG
122       Os::Position *position_;
123  #endif
124
125       // clustering algorithm core component
126       clustering_algo_t clustering_algo_;
127       /* End of stabilizing cluster vars */
128  };
129
130  // --------------------------------------------------------
131  // --------------------------------------------------------
132  // --------------------------------------------------------
133
134  wiselib::WiselibApplication<Os, ClusteringApplication>
         clustering_app;
135  // --------------------------------------------------------
136  void application_main( Os::AppMainParameter& value )
137  {
138       clustering_app.init( value );
```

```
139 }
```

Appendix B

Algorithm source code

Listing B.1: Source code of one-to-one highways algorithm.

```
1  /** This file is part of the generic algorithm        **
2  ** library Wiselib.                                   **
3  ** Copyright (C) 2008,2009 by the Wisebed             **
4  ** (www.wisebed.eu) project.                          **
5  **                                                    **
6  ** The Wiselib is free software: you can redistribute **
7  ** it and/or modify it under the terms of the GNU     **
8  ** Lesser General Public License as published by the  **
9  ** Free Software Foundation, either version 3 of the  **
10 ** License, or (at your option) any later version.    **
11 **                                                    **
12 ** The Wiselib is distributed in the hope that it     **
13 ** will be useful, but WITHOUT ANY WARRANTY; without  **
14 ** even the implied warranty of MERCHANTABILITY or    **
15 ** FITNESS FOR A PARTICULAR PURPOSE.  See the GNU     **
16 ** Lesser General Public License for more details.    **
17 **                                                    **
18 ** You should have received a copy of the GNU Lesser  **
19 ** General Public                                     **
20 ** License along with the Wiselib.                    **
21 ** If not, see <http://www.gnu.org/licenses/>.        **
22 ******************************************************/
23
```

```
24 #ifndef __ALGORITHMS_CLUSTER_HIGHWAY_CLUSTER_H__
25 #define __ALGORITHMS_CLUSTER_HIGHWAY_CLUSTER_H__
26
27
28 #include "algorithms/neighbor_discovery/echo.h"
29 #include "algorithms/cluster/clustering_types.h"
30 #include "util/pstl/vector_static.h"
31 #include "util/pstl/pair.h"
32 #include "util/pstl/map_static_vector.h"
33 #include "internal_interface/routing_table/
      routing_table_static_array.h"
34 #include "util/delegates/delegate.hpp"
35
36 #include "algorithms/cluster/fronts/fronts_core.h"
37 #include "algorithms/cluster/modules/chd/attr_chd.h"
38 #include "algorithms/cluster/modules/it/fronts_it.h"
39 #include "algorithms/cluster/modules/jd/bfs_jd.h"
40
41 // Levels of debug info:
42 //   HIGHWAY_METHOD_DEBUG: Adds a Method called notice
43 //     at the beginning of each method.
44 //   HIGHWAY_MSG_RECV_DEBUG: Adds all sorts of debug
45 //     messages for the tracking of sending messages at
46 //     Highway level.
47 //   VISOR_DEBUG: Adds debug messages that can be
48 //     processed by the Python visor application to
49 //     generate a picture of the final status of the WSN.
50 //   HIGHWAY_DEBUG: Most general debug messages.
51
52 //#define HIGHWAY_DEBUG
53 //#define HWY_DEBUG
54 //#define HWY_SEND_DEBUG
55 //#define HIGHWAY_METHOD_DEBUG
56 //#define HIGHWAY_MSG_RECV_DEBUG
57 //#define VISOR_DEBUG
58 #define CTI_VISOR
```

```
59 //#define TRACK_SEND_MSG
60 #ifdef ISENSE_APP
61 #define SEND_OVERHEAD 9
62 #else
63 #define SEND_OVERHEAD 17
64 #endif
65 //#define STABLE
66
67 namespace wiselib{
68 template<typename OsModel_P,
69      typename RoutingTable_P,
70      typename Cluster_P = wiselib::FrontsCore<OsModel_P,
         typename OsModel_P::TxRadio, wiselib::
         AtributeClusterHeadDecision<OsModel_P, typename
         OsModel_P::TxRadio>, wiselib::BfsJoinDecision<
         OsModel_P, typename OsModel_P::TxRadio>, wiselib::
         FrontsIterator<OsModel_P, typename OsModel_P::
         TxRadio> >,
71      typename Neighbor_P = wiselib::Echo<OsModel_P, typename
          OsModel_P::TxRadio, typename OsModel_P::Timer,
         typename OsModel_P::Debug>,
72      uint16_t MAX_CLUSTERS = 8>
73 class HighwayCluster
74 {
75 public:
76
77       // OS modules.
78      typedef OsModel_P OsModel;
79      typedef typename OsModel::Rand Rand;
80      typedef typename OsModel::TxRadio Radio;
81      typedef typename OsModel::Timer Timer;
82      typedef typename OsModel::Clock Clock;
83      typedef typename OsModel::Debug Debug;
84      typedef typename OsModel::TxRadio TxRadio;
85
86      // Type definitions.
```

```
87      typedef wiselib::AtributeClusterHeadDecision<OsModel,
           TxRadio> CHD_t;
88      typedef wiselib::BfsJoinDecision<OsModel, TxRadio> JD_t
           ;
89      typedef wiselib::FrontsIterator<OsModel, TxRadio> IT_t;
90
91      // Type definition of the used Templates.
92      typedef RoutingTable_P RoutingTable;
93      typedef Cluster_P Cluster;
94      typedef Neighbor_P Neighbor;
95      typedef typename RoutingTable::iterator
           routing_iterator;
96
97      typedef HighwayCluster<OsModel, RoutingTable, Cluster,
           Neighbor, MAX_CLUSTERS> self_type;
98      typedef wiselib::Echo<OsModel, TxRadio, Timer, Debug>
           nb_t;
99      typedef self_type* self_pointer_t;
100
101     // Basic types definition.
102     typedef typename Radio::node_id_t node_id_t;
103     typedef typename Radio::size_t size_t;
104     typedef typename Radio::block_data_t block_data_t;
105     typedef typename Timer::millis_t millis_t;
106
107     // Type definition for the receive callback.
108     typedef delegate3<void, node_id_t, size_t, block_data_t
           *> highway_delegate_t;
109
110     // Type definition for the special data structures
111     // of the highway.
112     typedef wiselib::pair<uint8_t, int8_t> hops_ack;
113     typedef wiselib::pair<node_id_t, node_id_t>
           source_target;
114     typedef wiselib::pair<source_target, hops_ack> entry;
115     typedef wiselib::MapStaticVector<OsModel, node_id_t,
```

```
        entry, MAX_CLUSTERS> HighwayTable;
116     typedef HighwayTable PortsQueue;
117     typedef wiselib::vector_static<OsModel, node_id_t,
        MAX_CLUSTERS> Node_vect;
118
119     // Type definition for the special types iterators.
120     typedef typename HighwayTable::iterator
        highway_iterator;
121
122     // Return types definition.
123     enum ErrorCodes {
124          SUCCESS = OsModel::SUCCESS,
125          ERR_UNSPEC = OsModel::ERR_UNSPEC
126     };
127
128     // Possible highway message ids.
129     enum msg_id {
130          CANDIDACY = 30,
131          PORT_REQ = 31,
132          PORT_REQ2 = 32,
133          PORT_ACK = 33,
134          PORT_ACK2 = 34,
135          PORT_NACK = 35,
136          PORT_NACK2 = 36,
137          SEND = 37,
138          SEND2 = 38,
139          ACK = 39,
140          ACK2 = 40
141     };
142
143     // --------------------------------------------------
144     // Public method declaration.                      |
145     // --------------------------------------------------
146
147     /** Constructor */
148     HighwayCluster():
```

```
149       radio_ ( 0 ),
150       timer_ ( 0 ),
151       clock_ ( 0 ),
152       debug_ ( 0 ),
153       rand_ ( 0 ),
154       cluster_ ( 0 ),
155       discovery_time_ ( 5000 ),
156       disc_timer_set_(false),
157       cand_timer_set_(false),
158       max_acks_(15),
159       reg_callback_ ( false ),
160       enabled_(false)
161       {
162       };
163
164       /** Destructor */
165       ~HighwayCluster(){};
166
167
168       /** Initialization method.
169        * @brief Sets the templated classes into pointers
170        *        and initializes the neighborhood discovery
171        *        module.
172        */
173       int init( TxRadio& tx_radio, Timer& timer, Clock& clock
             , Debug& debug, Rand& rand, Cluster& cluster,
             Neighbor& neighbor );
174
175       /** Highway enabling method.
176        * @brief Enables underlying modules and registers
177        * their callbacks.
178        */
179       void enable( void );
180
181       /** Highway sending method.
182        * @brief sends the data to the receiver cluster
```

```
183         *        head.
184         * @param receiver The cluster id of destination.
185         * @param len The length of the data to send.
186         * @param data The pointer to the data to send.
187         */
188        void send( node_id_t receiver, size_t len, block_data_t
                *data );
189
190        /** Cluster neighbors listing.
191         * @brief Gives a vector of clusters that are
192         *        neighbors to the current one.
193         * @return An empty vector if not called in the
194         *         cluster leader, a vector of the one hop
195         *         cluster ids otherwise.
196         */
197        //Node_vect cluster_neighbors();
198        void cluster_neighbors(Node_vect * neighbor);
199
200        /** Highway receive callback registering.
201         * @param obj_pnt An object with a method matching
202         *        the receive signature.
203         */
204        template<class T, void (T::*TMethod)(node_id_t, size_t,
                block_data_t*)>
205        uint8_t hwy_reg_recv_callback(T *obj_pnt) {
206             hwy_recv_callback_ = highway_delegate_t::template
                    from_method<T, TMethod > ( obj_pnt );
207             reg_callback_ = true;
208             return 0;
209        }
210
211        /** Highway receive callback unregistering. */
212        void unreg_hwy_recv_callback() {
213             hwy_recv_callback_ = highway_delegate_t();
214             reg_callback_ = false;
215        }
```

```
216
217       inline void disable(void) {
218            // Unregister the callback
219            radio().unreg_recv_callback(radio_callback_id_);
220            cluster().disable();
221            enabled_ = false;
222  #ifdef CTI_VISOR
223            debug().debug( "HWY_SHUT" );
224  #endif
225       }
226       ;
227
228       // -----------------------------------------------
229       // Setters                                       |
230       // -----------------------------------------------
231
232       /** Sets discovery time.
233        * @param t Time in milliseconds to set as
234        *          discovery_time_.
235        */
236       inline void set_discovery_time( millis_t t ) {
237            discovery_time_ = t;
238       };
239
240       /** Sets max acks.
241        * @param t Time in milliseconds to set as
242        *          discovery_time_.
243        */
244       inline void set_max_acks( uint8_t m ) {
245            max_acks_ = m;
246       };
247
248  private:
249       // Typenaming the underlying modules.
250       typename Radio::self_pointer_t radio_;
251       typename Timer::self_pointer_t timer_;
```

```
252         typename Clock::self_pointer_t clock_;
253         typename Debug::self_pointer_t debug_;
254         typename Rand::self_pointer_t rand_;
255         typename Cluster::self_type* cluster_;
256         typename Neighbor::self_t* neighbor_;
257
258         // Highway control message.
259         struct msg_highway {
260             uint8_t msg_id, hops;
261             node_id_t source, target, sid_source, sid_target;
262         };
263
264         enum msg_highway_size {
265             HWY_MSG_SIZE = sizeof( uint8_t ) + sizeof( uint8_t
                    ) + sizeof( node_id_t ) + sizeof( node_id_t )
                   + sizeof( node_id_t ) + sizeof( node_id_t )
266         };
267
268         inline void set_msg_highway( uint8_t * data, uint8_t
               msg_id, uint8_t hops, node_id_t source, node_id_t
               target, node_id_t sid_source, node_id_t sid_target )
269         {
270             int idx = 0;
271             write<OsModel, block_data_t, uint8_t>( data + idx,
                    msg_id );
272             idx += sizeof( uint8_t );
273             write<OsModel, block_data_t, uint8_t>( data + idx,
                    hops );
274             idx += sizeof( uint8_t );
275             write<OsModel, block_data_t, node_id_t>( data +
                   idx, source );
276             idx += sizeof( node_id_t );
277             write<OsModel, block_data_t, node_id_t>( data +
                   idx, target );
278             idx += sizeof( node_id_t );
279             write<OsModel, block_data_t, node_id_t>( data +
```

```
                idx, sid_source );
280             idx += sizeof( node_id_t );
281             write<OsModel, block_data_t, node_id_t>( data +
                    idx, sid_target );
282         }
283
284         inline void get_msg_highway( msg_highway * msg, uint8_t
                * data )
285         {
286             int idx = 0;
287             msg->msg_id = read<OsModel, block_data_t, uint8_t
                    >( data + idx );
288             idx += sizeof( uint8_t );
289             msg->hops = read<OsModel, block_data_t, uint8_t>(
                    data + idx );
290             idx += sizeof( uint8_t );
291             msg->source = read<OsModel, block_data_t,
                    node_id_t>( data + idx );
292             idx += sizeof( node_id_t );
293             msg->target = read<OsModel, block_data_t,
                    node_id_t>( data + idx );
294             idx += sizeof( node_id_t );
295             msg->sid_source = read<OsModel, block_data_t,
                    node_id_t>( data + idx );
296             idx += sizeof( node_id_t );
297             msg->sid_target = read<OsModel, block_data_t,
                    node_id_t>( data + idx );
298         }
299
300         // -------------------------------------------------
301         // Private variables declaration.                  |
302         // -------------------------------------------------
303
304         /** @brief Message used for control of the highways. */
305         msg_highway msg_highway_;
306
```

```
307       /** @brief Time allocated for the neighborhood
308        *         discovery module to do the initial survey */
309       millis_t discovery_time_;
310
311       /** @brief Checks if there is a discovery timeout
312        *         going on. */
313       bool disc_timer_set_;
314
315       /** @brief Checks if there is a candidacies timeout
316        *         going on. */
317       bool cand_timer_set_;
318
319       /** @brief Highway message received callback to
320        *         processor. */
321       highway_delegate_t hwy_recv_callback_;
322
323
324       /** @brief Used for the intra cluster routing. */
325       RoutingTable routing_table_;
326
327       /** @brief Cluster leader highway routing. */
328       HighwayTable highway_table_;
329
330       /** @brief Queue of port candidates. */
331       PortsQueue ports_queue_;
332
333       /** @brief Max size buffer for sending the highway
334        *         level messages. */
335       block_data_t buffer_[Radio::MAX_MESSAGE_LENGTH];
336
337       /** @brief In negative, minimum number of acks
338        *         not received */
339       int8_t max_acks_;
340       int radio_callback_id_;
341       bool reg_callback_;
342       bool enabled_;
```

```
343     node_id_t clus_head_;
344
345
346     /** @brief Clustering helper modules declaration. */
347     CHD_t CHD_;
348     JD_t JD_;
349     IT_t IT_;
350
351
352     // -------------------------------------------------
353     // Private method declaration.                     |
354     // -------------------------------------------------
355
356     /** Highway sending method.
357      * @brief sends the data to the receiver cluster head.
358      * @param send_ack True if sending message, false
359      *                 when acking.
360      * @param receiver The cluster id of destination.
361      * @param len The length of the data to send.
362      * @param data The pointer to the data to send.
363      */
364     void send( bool send_ack, node_id_t receiver, size_t
        len, block_data_t *data );
365
366     /**
367      * @return the Radio module.
368      */
369     Radio& radio() {
370         return *radio_;
371     }
372
373     /**
374      * @return the Timer module.
375      */
376     Timer& timer() {
377         return *timer_;
```

```
378         }
379
380         /**
381          * @return the Clock module.
382          */
383         Clock& clock() {
384              return *clock_;
385         }
386
387         /**
388          * @return the Debug module.
389          */
390         Debug& debug() {
391              return *debug_;
392         }
393
394         /**
395          * @return the Cluster module.
396          */
397         Cluster& cluster() {
398              return *cluster_;
399         }
400
401         /**
402          * @return the Neighbor module.
403          */
404         Neighbor& neighbor() {
405              return *neighbor_;
406         }
407
408         /** Clustering events callback.
409          * @brief Gets the cluster event that should start
410          *        construction.
411          * @param state The event generated by the Cluster
412          *              module.
413          */
```

```
414       void cluster_callback(int state);
415
416       void clean_highways( bool all, bool notify );
417
418       /** Highway cluster discovery.
419        * @brief Piggyback information on the neighborhood
420        *        discovery module and register its callback.
421        */
422       void cluster_discovery( void );
423
424       /** Neighbor callback registering.
425        * @brief Receive the neighborhood discovery
426        *        piggybacked data accordint to event.
427        * @param event The neighborhood discovery event
428        *              that issues this callback.
429        * @param from The origin of the transmission that
430        *             issues this callback.
431        * @param len The length of the piggybacked data.
432        * @param data The piggybacked data.
433        */
434       void neighbor_callback( uint8_t event, node_id_t from,
          uint8_t len, uint8_t* data);
435
436       /** Discovery timeout.
437        * @brief On discovery timeout the behavior is as
438        *        follows: The other nodes send a CANDIDACY
439        *        message to the cluster leader. Waits for
440        *        a work period to update the information.
441        */
442       void discovery_timeout( void *userdata );
443
444       /** Candidacies timeout.
445        * @brief On candidacies timeout the behavior is as
446        *        follows: The cluster leader picks some
447        *        candidates and starts to negotiate ports.
448        *        Waits for a work period to reconstruct.
```

```
449         */
450        void candidacies_timeout( void *userdata );
451
452        /** Radio receive callback.
453         * @brief calls the specific method to process the
454         *        several kinds of highway_msg.
455         * @param from The immediate sender of the message.
456         * @param len Length of the data sent.
457         * @param data Pointer to the data sent.
458         */
459        void receive( node_id_t from, size_t len, block_data_t
             *data );
460
461        /** Messages going towards own cluster leader.
462         * @brief Propagates the messages to the cluster
463         *        leader.
464         * @param from The immediate sender of the message.
465         * @param len Length of the data sent.
466         * @param data Pointer to the data sent.
467         */
468        void send_to_leader( node_id_t from, size_t len,
             block_data_t *data );
469
470        /** Messages going towards another cluster.
471         * @brief Propagates the messages to the target
472         *        cluster.
473         * @param from The immediate sender of the message.
474         * @param len Length of the data sent.
475         * @param data Pointer to the data sent.
476         */
477        void send_away( node_id_t from, size_t len,
             block_data_t *data );
478
479        /** Message type SEND processing.
480         * @brief Propagates the encapsulated payload to the
481         *        target cluster leader.
```

```
482         * @param from The immediate sender of the message.
483         * @param len Length of the data sent.
484         * @param data Pointer to the data sent.
485         */
486        void process_send( node_id_t from, size_t len,
              block_data_t *data );
487
488        /** Received message processing by the cluster leader.
489         * @brief Processes the message and updates data
490         *        structures.
491         * @param from The immediate sender of the message.
492         * @param len Length of the data sent.
493         * @param data Pointer to the data sent.
494         */
495        void cluster_head_work( node_id_t from, size_t len,
              block_data_t *data );
496
497
498  }; // End of class.
499
500  // ------------------------------------------------------
501  // Start of the method code: PUBLIC METHODS             |
502  // ------------------------------------------------------
503  template<typename OsModel_P,
504           typename RoutingTable_P,
505           typename Cluster_P,
506           typename Neighbor_P,
507           uint16_t MAX_CLUSTERS>
508  inline int
509  HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
510  init( TxRadio& tx_radio, Timer& timer, Clock& clock, Debug&
       debug, Rand& rand, Cluster& cluster, Neighbor& neighbor )
        {
511       radio_ = &tx_radio;
512       timer_ = &timer;
```

```
513       clock_ = &clock;
514       debug_ = &debug;
515       rand_ = &rand;
516       cluster_ = &cluster;
517       neighbor_ = &neighbor;
518
519       // Initialize the neighborhood discovery module.
520       neighbor_->init( tx_radio, *clock_, *timer_, *debug_ );
521       if ( neighbor_->register_payload_space( HWY_N ) !=0 )
522       {
523 #ifdef HIGHWAY_DEBUG
524             debug().debug( "Error registering payload space" )
                  ;
525 #endif
526       }
527
528       clus_head_ = 0;
529
530       // Stabilizing cluster initialization.
531       // set the HeadDecision Module
532       cluster_->set_cluster_head_decision( CHD_ );
533       // set the JoinDecision Module
534       cluster_->set_join_decision( JD_ );
535       // set the Iterator Module
536       cluster_->set_iterator( IT_ );
537       cluster_->init( *radio_, *timer_, *debug_, *rand_, *
             neighbor_ );
538
539       //cluster_->set_maxhops( 1 );
540       cluster_->set_maxhops( 2 );
541
542       return SUCCESS;
543 }
544
545 // -----------------------------------------------------
546
```

```
547 template<typename OsModel_P,
548          typename RoutingTable_P,
549          typename Cluster_P,
550          typename Neighbor_P,
551          uint16_t MAX_CLUSTERS>
552 void
553 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
554 enable( void )
555 {
556 #ifdef HIGHWAY_METHOD_DEBUG
557      debug().debug( "@@ %x METHOD CALLED: enable()\n", radio
           ().id() );
558 #endif
559      enabled_ = true;
560      // Enabling and registering radio
561      radio().enable_radio();
562      radio_callback_id_ = radio().template reg_recv_callback
           <self_type, &self_type::receive>( this );
563
564      // Enabling neighborhood and registering cluster
565      cluster().enable();
566      cluster().template reg_state_changed_callback<self_type
           , &self_type::cluster_callback > ( this );
567      neighbor().enable();
568
569      neighbor().template reg_event_callback<HighwayCluster,&
           HighwayCluster::neighbor_callback>( HWY_N, nb_t::
           NEW_PAYLOAD_BIDI, this );
570
571 #ifdef HIGHWAY_METHOD_DEBUG
572      debug().debug( "@@ %x METHOD ENDED: enable()\n", radio
           ().id() );
573 #endif
574 }
575
```

```
576 // -------------------------------------------------------
577
578 template<typename OsModel_P,
579          typename RoutingTable_P,
580          typename Cluster_P,
581          typename Neighbor_P,
582          uint16_t MAX_CLUSTERS>
583 void
584 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
585 send( node_id_t destination, size_t len, block_data_t *data
       )
586 {
587      if(!enabled_)
588           return;
589 #ifdef HIGHWAY_METHOD_DEBUG
590      debug().debug( "@@ %x METHOD CALLED: send()\n", radio()
           .id() );
591 #endif
592
593      send(true, destination, len, data);
594
595 #ifdef HIGHWAY_METHOD_DEBUG
596      debug().debug( "@@ %x METHOD ENDED: send()\n", radio().
           id() );
597 #endif
598 }
599
600 // -------------------------------------------------------
601
602 template<typename OsModel_P,
603          typename RoutingTable_P,
604          typename Cluster_P,
605          typename Neighbor_P,
606          uint16_t MAX_CLUSTERS>
607 void
```

```
608 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
609 cluster_neighbors( Node_vect * neighbors )
610 {
611      if(!enabled_)
612           return;
613 #ifdef HIGHWAY_METHOD_DEBUG
614      debug().debug( "@@ %x METHOD CALLED: cluster_neighbors
            ()\n", radio().id() );
615 #endif
616
617      neighbors->clear();
618      if ( not cluster().is_cluster_head() )
619          return;
620
621      clean_highways( false, true );
622
623      for ( highway_iterator it = highway_table_.begin(); it
           != highway_table_.end(); ++it )
624           neighbors->push_back( it->first );
625
626 #ifdef HIGHWAY_METHOD_DEBUG
627      debug().debug( "@@ %x METHOD ENDED: cluster_neighbors()
           \n", radio().id() );
628 #endif
629 }
630
631 // -----------------------------------------------------
632 // PRIVATE METHODS                                     |
633 // -----------------------------------------------------
634
635 template<typename OsModel_P,
636          typename RoutingTable_P,
637          typename Cluster_P,
638          typename Neighbor_P,
639          uint16_t MAX_CLUSTERS>
```

```
640 void
641 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
642 clean_highways( bool all, bool notify )
643 {
644      if(!enabled_)
645           return;
646      highway_iterator it = highway_table_.begin();
647      while( it != highway_table_.end() )
648      {
649           if( all || it->second.second.second > max_acks_ )
650           {
651                if(notify)
652                {
653 #ifdef CTI_VISOR
654                     debug().debug( "HWY_DEL; %x; %x; %x; %x"
                          , it->second.first.second, it->second
                          .first.first, it->first, clus_head_ )
                          ;
655 #endif
656                     // Create a CANDIDACY highway message
657                     // with: hops, port_source, port_target,
658                     // cluster_id_own, cluster_id_target.
659                     set_msg_highway( buffer_, PORT_NACK, it
                          ->second.second.first, it->second.
                          first.second, it->second.first.first,
                           it->first, clus_head_ );
660
661                     if( radio().id() == it->second.first.
                          first )
662                     {
663                       radio().send(it->second.first.
                             second, HWY_MSG_SIZE, buffer_);
664                     }
665                     else
666                     {
```

```
667                             routing_iterator rit =
                                   routing_table_.find( it->second.
                                   first.first );
668                             if( rit != routing_table_.end() )
669                                  radio().send(rit->second,
                                        HWY_MSG_SIZE, buffer_);
670                         }
671                     }
672                     highway_table_.erase(it->first);
673                     it = highway_table_.begin();
674                }
675                else
676                {
677                     ++it;
678                }
679           }
680      }
681
682      // -------------------------------------------------------
683
684      template<typename OsModel_P,
685               typename RoutingTable_P,
686               typename Cluster_P,
687               typename Neighbor_P,
688               uint16_t MAX_CLUSTERS>
689      void
690      HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
           Neighbor_P, MAX_CLUSTERS>::
691      cluster_callback( int state )
692      {
693           if(!enabled_)
694                return;
695      #ifdef HIGHWAY_METHOD_DEBUG
696           debug().debug( "@@ %x METHOD_CALLED: cluster_callback()
                \n", radio().id() );
697      #endif
```

```
698
699 #ifdef VISOR_DEBUG
700       debug().debug( "+%d#%d#%d#1\n", radio().id(), cluster()
              .cluster_id(), cluster().is_cluster_head() );
701 #endif
702
703       // Check if it is a real change.
704       if( clus_head_ != cluster().cluster_id() )
705       {
706 #ifdef CTI_VISOR
707           debug().debug( "HWY_CLUS; %x; %d", cluster().
                cluster_id(), cluster().parent() );
708 #endif
709           clean_highways( true, clus_head_ == radio().id() )
                ;
710           ports_queue_.clear();
711           //routing_table_.clear();
712           clus_head_ = cluster().cluster_id();
713           cluster_discovery();
714       }
715
716 #ifdef HIGHWAY_METHOD_DEBUG
717       debug().debug( "@@ %x METHOD_ENDED: clustering_callback
              ()\n", radio().id() );
718 #endif
719 }
720
721 // -----------------------------------------------------
722
723 template<typename OsModel_P,
724          typename RoutingTable_P,
725          typename Cluster_P,
726          typename Neighbor_P,
727          uint16_t MAX_CLUSTERS>
728 void
729 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
```

```
    Neighbor_P, MAX_CLUSTERS>::
730 cluster_discovery( void )
731 {
732     if(!enabled_)
733         return;
734 #ifdef HIGHWAY_METHOD_DEBUG
735     debug().debug( "@@ %x METHOD CALLED: cluster_discovery
        ()\n", radio().id() );
736 #endif
737
738     // Add the piggybacking information to the
739     // neighborhood discovery module.
740     node_id_t id = cluster().cluster_id();
741     uint8_t length;
742
743     // Adapt cluster_id to the node_id_t size of
744     // the plattform.
745 #ifdef ISENSE_APP
746     uint8_t sid_buf[3];
747     sid_buf[0] = id & 0xFF;
748     sid_buf[1] = id >> 8;
749     sid_buf[2] = cluster().hops()+1;
750     length = 3;
751 #else
752     uint8_t sid_buf[5];
753     sid_buf[0] = id & 0xFF;
754     sid_buf[1] = ( id >> 8 ) & 0xFF;
755     sid_buf[2] = ( id >> 16 ) & 0xFF;
756     sid_buf[3] = ( id >> 24 ) & 0xFF;
757     sid_buf[4] = cluster().hops()+1;
758     length = 5;
759 #endif
760
761     // Register and add the payload space to the
762     // neighborhood discovery module.
763
```

```
764       if(neighbor().set_payload( HWY_N, sid_buf, length )!=0
              )
765       {
766  #ifdef HWY_DEBUG
767            debug().debug( "Error setting payload" );
768  #endif
769       }
770       else
771       {
772  #ifdef HWY_DEBUG
773            debug().debug( "Registering neighborhood" );
774  #endif
775            if( not disc_timer_set_ )
776            {
777                 disc_timer_set_ = true;
778                 timer().template set_timer<self_type, &
                        self_type::discovery_timeout>(
                        discovery_time_ , this, (void *) 0 );
779            }
780       }
781  #ifdef HIGHWAY_METHOD_DEBUG
782       debug().debug( "@@ %x METHOD ENDED: cluster_discovery()
              \n", radio().id() );
783  #endif
784  }
785
786  // -----------------------------------------------------
787
788  template<typename OsModel_P,
789           typename RoutingTable_P,
790           typename Cluster_P,
791           typename Neighbor_P,
792           uint16_t MAX_CLUSTERS>
793  void
794  HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
```

```
795 neighbor_callback( uint8_t event, node_id_t from, uint8_t
       len, uint8_t* data)
796 {
797       if(!enabled_)
798             return;
799 #ifdef HIGHWAY_METHOD_DEBUG
800       debug().debug("@@ %x METHOD CALLED: neighbor_callback()
             \n", radio().id());
801 #endif
802
803       memcpy( buffer_, data, len );
804
805       node_id_t sid;
806       uint8_t hops;
807
808       // On payload event, process the data according
809       // to the plattform
810       if ( nb_t::NEW_PAYLOAD_BIDI == event ) {
811 #ifdef ISENSE_APP
812             sid = ( buffer_[1] << 8 ) | buffer_[0];
813             hops = buffer_[2];
814 #else
815             sid = ( buffer_[3] << 24 ) | ( buffer_[2] << 16 )
                   | ( buffer_[1] << 8 ) | buffer_[0];
816             hops = buffer_[4];
817 #endif
818
819             // If the message is from another cluster add
820             // it to the ports queue.
821             if ( cluster().cluster_id() != sid  && sid !=
                   radio().id() && ( cluster().is_cluster_head()
                   || cluster().hops() != 0 ) )
822             {
823 #ifdef HIGHWAY_DEBUG
824                   debug().debug( "%d NB_disc <-%d+%d-%x(%x)",
                         cluster().is_cluster_head(), cluster().
```

```
                    hops(), hops, from, sid );
825 #endif
826                 //Check if it is new and that it is better
827                 // than the current one
828                 highway_iterator it = ports_queue_.find( sid
                    );
829 #ifdef STABLE
830                 if( it == ports_queue_.end() || it->second.
                    second.first > hops )
831 #else
832                 if( true )
833 #endif
834                 {
835                     ports_queue_[sid] = entry( source_target
                        ( radio().id(), from ), hops_ack(
                        hops+cluster().hops(), 0 ));
836                 }
837                 else
838                 {
839 #ifdef HIGHWAY_DEBUG
840                     debug().debug( "%d Refused NB_disc <-%d
                        +%d-%x(%x)", cluster().
                        is_cluster_head(), cluster().hops(),
                        hops, from, sid );
841 #endif
842                     return;
843                 }
844
845                 if( cluster().is_cluster_head() )
846                 {
847                     if( not cand_timer_set_ )
848                     {
849                         cand_timer_set_ = true;
850                         timer().template set_timer<
                            self_type, &self_type::
                            candidacies_timeout>(
```

```
                                discovery_time_ , this, (void *)
                                 0 );
851                         }
852                     }
853                     else
854                     {
855                         if( not disc_timer_set_ )
856                         {
857                             disc_timer_set_ = true;
858                             timer().template set_timer<
                                    self_type, &self_type::
                                    discovery_timeout>(
                                    discovery_time_ , this, (void *)
                                     0 );
859                         }
860                     }
861                 }
862
863         }
864 #ifdef HIGHWAY_METHOD_DEBUG
865         debug().debug("@@ %x METHOD ENDED: neighbor_callback()\
              n", radio().id());
866 #endif
867 }
868
869 // -----------------------------------------------------
870
871 template<typename OsModel_P,
872     typename RoutingTable_P,
873     typename Cluster_P,
874     typename Neighbor_P,
875     uint16_t MAX_CLUSTERS>
876 void
877 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
878 discovery_timeout( void *userdata )
```

```
879 {
880      if(!enabled_)
881           return;
882 #ifdef HIGHWAY_METHOD_DEBUG
883 debug().debug( "@@ %x METHOD CALLED: discovery_timeout()\n",
        radio().id());
884 #endif
885
886 highway_iterator pit, hit;
887 for ( pit = ports_queue_.begin(); pit != ports_queue_.end();
        ++pit )
888 {
889      //Check if the port candidate is already set as highway
890      hit = highway_table_.find( pit->first );
891      if( hit != highway_table_.end() && hit->second.first.
           first == pit->second.first.first && hit->second.
           first.second == pit->second.first.second )
892      {
893 #ifdef HIGHWAY_DEBUG
894           debug().debug( "There was already highway for %x",
                pit->first );
895 #endif
896           continue;
897      }
898      // Create a CANDIDACY highway message with: hops,
899      // port_source, port_target, cluster_id_own,
900      // cluster_id_target.
901      set_msg_highway( buffer_, CANDIDACY, pit->second.second
           .first, radio().id(), pit->second.first.second,
           cluster().cluster_id(), pit->first );
902
903      if( cluster().is_cluster_head() )
904      {
905           cluster_head_work( radio().id(), HWY_MSG_SIZE,
                buffer_ );
906      }
```

```
907       else
908       {
909            // Send it to the parent.
910            radio().send( cluster().parent(), HWY_MSG_SIZE,
                  buffer_ );
911       }
912  }
913  // OPTIONAL: Test with not flushing when cluster head,
914  // and locking the nb_bidi writing to leader ports_queue_
915  // when on candidacy timeout.
916  ports_queue_.clear();
917  disc_timer_set_ = false;
918  #ifdef HIGHWAY_METHOD_DEBUG
919  debug().debug( "@@ %x METHOD ENDED: discovery_timeout()\n",
       radio().id() );
920  #endif
921  }
922
923  //
       ------------------------------------------------------------------------


924
925  template<typename OsModel_P,
926      typename RoutingTable_P,
927      typename Cluster_P,
928      typename Neighbor_P,
929      uint16_t MAX_CLUSTERS>
930  void
931  HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
932  candidacies_timeout( void *userdata )
933  {
934       if(!enabled_)
935            return;
936  #ifdef HIGHWAY_METHOD_DEBUG
937  debug().debug( "@@ %x METHOD STARTED: candidacies_timeout()\
```

```
    n", radio().id() );
938 #endif
939 if( !cluster().is_cluster_head() )
940     return;
941
942 highway_iterator pit, hit;
943 for ( pit = ports_queue_.begin(); pit != ports_queue_.end();
        ++pit )
944 {
945     // Check that it is not the same highway
946     hit = highway_table_.find( pit->first );
947     if( hit != highway_table_.end() && ( hit->second.second
          .second <= max_acks_ || ( pit->second.first.first ==
           hit->second.first.first && pit->second.first.second
           == hit->second.first.second ) ) )
948         continue;
949
950     // Create a PORT_REQ highway message with: hops,
951     // port_source, port_target, cluster_id_own,
952     // cluster_id_target.
953 #ifdef HWY_DEBUG
954     debug().debug( "Negotiating %x; %x; %x; %x; %d", pit->
          second.first.first, pit->second.first.second,
          cluster().cluster_id(), pit->first, pit->second.
          second.first );
955 #endif
956     set_msg_highway( buffer_, PORT_REQ, pit->second.second.
          first, pit->second.first.first, pit->second.first.
          second, cluster().cluster_id(), pit->first );
957     send_away( radio().id(), HWY_MSG_SIZE, buffer_ );
958 }
959
960 //After processing all the ports, we flush the queue.
961 ports_queue_.clear();
962 cand_timer_set_ = false;
963 #ifdef HIGHWAY_METHOD_DEBUG
```

```
964       debug().debug( "@@ %x METHOD ENDED: candidacies_timeout
              ()\n", radio().id() );
965   #endif
966   }
967
968   // ----------------------------------------------------
969
970   template<typename OsModel_P,
971            typename RoutingTable_P,
972            typename Cluster_P,
973            typename Neighbor_P,
974            uint16_t MAX_CLUSTERS>
975   void
976   HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
          Neighbor_P, MAX_CLUSTERS>::
977   receive( node_id_t from, size_t len, block_data_t *data )
978   {
979       if(!enabled_)
980            return;
981   #ifdef HIGHWAY_METHOD_DEBUG
982       debug().debug("@@ %d METHOD_CALLED: receive()\n", radio
              ().id());
983   #endif
984
985       // Ignore if heard oneself's message.
986       if ( from == radio().id() )
987            return;
988
989       memcpy( &buffer_, data, len);
990
991   #ifdef HIGHWAY_MSG_RECV_DEBUG
992       if ( buffer_[0] == CANDIDACY ) debug().debug( "@@ %x(%d
              )<- %x: MSG_RECEIVED: CANDIDACY\n", radio().id(),
              cluster().is_cluster_head(), from );
993       else if ( buffer_[0] == PORT_REQ ) debug().debug( "@@ %
              x(%d)<- %x: MSG_RECEIVED: PORT_REQ\n", radio().id(),
```

```
            cluster().is_cluster_head(), from );
994       else if ( buffer_[0] == PORT_REQ2 ) debug().debug( "@@
             %x(%d)<- %x: MSG_RECEIVED: PORT_REQ2\n", radio().id
             (), cluster().is_cluster_head(), from );
995       else if ( buffer_[0] == PORT_ACK ) debug().debug( "@@ %
             x(%d)<- %x: MSG_RECEIVED: PORT_ACK\n", radio().id(),
              cluster().is_cluster_head(), from );
996       else if ( buffer_[0] == PORT_ACK2 ) debug().debug( "@@
             %x(%d)<- %x: MSG_RECEIVED: PORT_ACK2\n", radio().id
             (), cluster().is_cluster_head(), from );
997       else if ( buffer_[0] == PORT_NACK ) debug().debug( "@@
             %x(%d)<- %x: MSG_RECEIVED: PORT_NACK\n", radio().id
             (), cluster().is_cluster_head(), from );
998       else if ( buffer_[0] == PORT_NACK2 ) debug().debug( "@@
              %x(%d)<- %x: MSG_RECEIVED: PORT_NACK2\n", radio().
             id(), cluster().is_cluster_head(), from );
999       else if ( buffer_[0] == SEND ) debug().debug( "@@ %x(%d
             )<- %x: MSG_RECEIVED: SEND\n", radio().id(), cluster
             ().is_cluster_head(), from );
1000      else if ( buffer_[0] == SEND2 ) debug().debug( "@@ %d(%
             d)<- %x: MSG_RECEIVED: SEND2\n", radio().id(),
             cluster().is_cluster_head(), from );
1001  #endif
1002
1003  #ifdef CTI_VISOR
1004        if ( buffer_[0] == CANDIDACY ) debug().debug( "HWY_MSG;
              CAND; %x; %x", radio().id(), from );
1005        else if ( buffer_[0] == PORT_REQ  or buffer_[0] ==
             PORT_REQ2 ) debug().debug( "HWY_MSG; REQ; %x; %x, %d
             ", radio().id(), from, cluster().is_cluster_head() )
             ;
1006        else if ( buffer_[0] == PORT_ACK  or buffer_[0] ==
             PORT_ACK2 ) debug().debug( "HWY_MSG; PACK; %x; %x, %
             d", radio().id(), from, cluster().is_cluster_head()
             );
1007        else if ( buffer_[0] == PORT_NACK  or buffer_[0] ==
```

```
            PORT_NACK2 ) debug().debug( "HWY_MSG; PNACK; %x; %x"
            , radio().id(), from );
1008        else if ( buffer_[0] == SEND  or buffer_[0] == SEND2 )
            debug().debug( "HWY_MSG; SEND; %x; %x; ", radio().id
            (), from );
1009        else if ( buffer_[0] == ACK  or buffer_[0] == ACK2 )
            debug().debug( "HWY_MSG; ACK; %x; %x", radio().id(),
             from );
1010    #endif
1011
1012        // Messages travelling to the current node cluster
1013        // leader are processed in send_to_leader
1014        if ( buffer_[0] == CANDIDACY or buffer_[0] == PORT_REQ2
             or buffer_[0] == PORT_ACK2 or buffer_[0] ==
            PORT_NACK2 or buffer_[0] == SEND2 or buffer_[0] ==
            ACK2 )
1015        {
1016            send_to_leader( from, len, buffer_ );
1017        }
1018        // Construction messages travelling outwards
1019        else if ( buffer_[0] == PORT_REQ or buffer_[0] ==
            PORT_ACK or buffer_[0] == PORT_NACK )
1020        {
1021            send_away( from, len, buffer_ );
1022        }
1023        // Data messages travelling outwards.
1024        else if ( buffer_[0] == SEND or buffer_[0] == ACK )
1025        {
1026            process_send( from, len, buffer_ );
1027        }
1028    }
1029
1030    // -----------------------------------------------------
1031
1032    template<typename OsModel_P,
1033             typename RoutingTable_P,
```

```
1034            typename Cluster_P,
1035            typename Neighbor_P,
1036            uint16_t MAX_CLUSTERS>
1037  void
1038  HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
        Neighbor_P, MAX_CLUSTERS>::
1039  send_to_leader( node_id_t from, size_t len, block_data_t *
        data )
1040  {
1041       if(!enabled_)
1042            return;
1043  #ifdef HIGHWAY_METHOD_DEBUG
1044       debug().debug( "@@ %x METHOD_CALLED: send_to_leader()\n
           ", radio().id()) ;
1045  #endif
1046       uint8_t type = data[0];
1047
1048  #ifdef HIGHWAY_MSG_RECV_DEBUG
1049       if( type == SEND2 )
1050       {
1051            debug().debug( "---------------SENDRECV2
                --------------\n" );
1052            for( int i = 0; i<len; ++i )
1053                 debug().debug( "Data[%d]: %d\n", i, data[i] )
                      ;
1054            debug().debug( "---------------/SENDRECV2
                --------------\n" );
1055       }
1056  #endif
1057
1058       get_msg_highway( &msg_highway_, data );
1059       routing_table_[msg_highway_.source] = from;
1060
1061       if( type == PORT_ACK2 )
1062       {
1063  #ifdef VISOR_DEBUG
```

```
1064            debug().debug( "$%d->%d$h\n", from, radio_->id() )
                    ;
1065  #endif
1066  #ifdef CTI_VISOR
1067            debug().debug( "HWY_EDGE; %x; %x; %x; %x; %x",
                    from, msg_highway_.source, msg_highway_.target,
                     msg_highway_.sid_source, msg_highway_.
                    sid_target );
1068  #endif
1069       }
1070
1071       if( cluster().is_cluster_head() )
1072       {
1073            cluster_head_work( from, len, data );
1074       }
1075       else
1076       {
1077            if(msg_highway_.sid_source != radio().id() )
1078                 radio().send( cluster().parent(), len, data )
                         ;
1079       }
1080  #ifdef HIGHWAY_METHOD_DEBUG
1081       debug().debug( "@@ %x METHOD_ENDED: send_to_leader()\n"
             , radio().id()) ;
1082  #endif
1083  }
1084
1085  // ------------------------------------------------------
1086
1087  template<typename OsModel_P,
1088           typename RoutingTable_P,
1089           typename Cluster_P,
1090           typename Neighbor_P,
1091           uint16_t MAX_CLUSTERS>
1092  void
1093  HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
```

```
    Neighbor_P, MAX_CLUSTERS>::
1094 send_away( node_id_t from, size_t len, block_data_t *data )
1095 {
1096      if(!enabled_)
1097           return;
1098 #ifdef HIGHWAY_METHOD_DEBUG
1099      debug().debug("@@ %x METHOD_CALLED: send_away()\n",
         radio().id());
1100 #endif
1101      node_id_t next;
1102
1103      get_msg_highway( &msg_highway_, data );
1104
1105      // Check if we are still in the cluster that
1106      // originated the request.
1107      if ( from == cluster().parent() || from == radio().id()
          )
1108      {
1109           // If the current node is not the port,
1110           // continue the way to the port.
1111           if( msg_highway_.source != radio().id() &&
              msg_highway_.target != radio().id() )
1112           {
1113                routing_iterator rit;
1114                if ( msg_highway_.msg_id == PORT_REQ )
1115                     rit = routing_table_.find( msg_highway_.
                        source );
1116                else if( msg_highway_.msg_id == PORT_ACK )
1117                {
1118 #ifdef CTI_VISOR
1119                     debug().debug( "HWY_EDGE; %x; %x; %x; %x
                        ; %x", from, msg_highway_.source,
                        msg_highway_.target, msg_highway_.
                        sid_source, msg_highway_.sid_target )
                        ;
1120 #endif
```

```
1121                    rit = routing_table_.find( msg_highway_.
                           target );
1122               }
1123               else
1124                    rit = routing_table_.find( msg_highway_.
                           target );
1125               if( rit == routing_table_.end() )
1126               {
1127  #ifdef HWY_DEBUG
1128                    debug().debug( "PITFALL2" );
1129  #endif
1130                    return;
1131               }
1132               next = rit->second;
1133  #ifdef HWY_DEBUG
1134               debug().debug( "Still same cluster, not yet
                      port, passing request to %x", next );
1135  #endif
1136          }
1137          // It is the port. Send the message to the other
1138          // cluster port and set the port to highway
1139          // status(if needed).
1140          else
1141          {
1142               if( msg_highway_.msg_id == PORT_REQ )
1143               {
1144                    next = msg_highway_.target;
1145                    if( next == radio().id() )
1146                    {
1147  #ifdef HWY_DEBUG
1148                         debug().debug( "Error_1" );
1149  #endif
1150                         return;
1151                    }
1152               }
1153               else
```

```
1154                 {
1155                      next = msg_highway_.source;
1156                      if( next == radio().id() )
1157                      {
1158 #ifdef HWY_DEBUG
1159                           debug().debug( "Error_2" );
1160 #endif
1161                           return;
1162                      }
1163                      if( msg_highway_.msg_id == PORT_ACK )
1164                      {
1165                           if( not cluster().is_cluster_head()
                                  )
1166                                highway_table_[msg_highway_.
                                   sid_source] = entry(
                                   source_target( msg_highway_
                                   .target, msg_highway_.
                                   source ), hops_ack(
                                   msg_highway_.hops, 0 ) );
1167 #ifdef CTI_VISOR
1168                           debug().debug( "HWY_EDGE; %x; %x; %
                              x; %x; %x", from, msg_highway_.
                              source, msg_highway_.target,
                              msg_highway_.sid_source,
                              msg_highway_.sid_target );
1169 #endif
1170 #ifdef VISOR_DEBUG
1171                           debug().debug("$%d->%d$h\n", from,
                              radio_->id());
1172 #endif
1173                      }
1174                 }
1175 #ifdef HWY_DEBUG
1176                 debug().debug( "Still same cluster, already
                    port, passing request to %x", msg_highway_
                    .target );
```

```
1177 #endif
1178           }
1179           radio().send( next, HWY_MSG_SIZE, data );
1180      }
1181      else // Create a "2" message.
1182      {
1183           // Make it into a "2" message
1184           if ( msg_highway_.msg_id == PORT_ACK )
1185           {
1186 #ifdef VISOR_DEBUG
1187                debug().debug( "+%d#%d#%d#1\n", radio().id(),
                         cluster().cluster_id(), cluster().
                        is_cluster_head() );
1188                debug().debug( "+%d#%d#%d#1\n", from,
                        msg_highway_.sid_target, cluster().
                        is_cluster_head() );
1189 #endif
1190 #ifdef CTI_VISOR
1191 //               debug().debug( "HWY_EDGE; %x; %x; %x; %x; %
       x", from, msg_highway_.source, msg_highway_.target,
       msg_highway_.sid_source, msg_highway_.sid_target );
1192 #endif
1193 #ifdef CTI_VISOR
1194                debug().debug( "HWY_PORTS; %x; %x", radio().
                        id(), from );
1195 #endif
1196                if( not cluster().is_cluster_head() )
1197                     highway_table_[msg_highway_.sid_target]
                             = entry( source_target( msg_highway_.
                             source, from ), hops_ack(
                             msg_highway_.hops, 0 ) );
1198           }
1199           msg_highway_.msg_id++;
1200
1201           set_msg_highway( buffer_, msg_highway_.msg_id,
                   msg_highway_.hops, msg_highway_.source,
```

```
                msg_highway_.target, msg_highway_.sid_source,
                msg_highway_.sid_target );
1202            if( cluster().is_cluster_head() )
1203                 cluster_head_work( from, len, buffer_ );
1204            else
1205                 radio().send( cluster().parent(),
                        HWY_MSG_SIZE, buffer_ );
1206       }
1207  #ifdef HIGHWAY_METHOD_DEBUG
1208       debug().debug("@@ %x METHOD_ENDED: send_away()\n",
             radio().id());
1209  #endif
1210  }
1211
1212  // -----------------------------------------------------
1213
1214  template<typename OsModel_P,
1215           typename RoutingTable_P,
1216           typename Cluster_P,
1217           typename Neighbor_P,
1218           uint16_t MAX_CLUSTERS>
1219  void
1220  HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
        Neighbor_P, MAX_CLUSTERS>::
1221  process_send( node_id_t from, size_t len, block_data_t *data
         )
1222  {
1223       if(!enabled_)
1224            return;
1225  #ifdef HIGHWAY_METHOD_DEBUG
1226       debug().debug("@@ %x METHOD_CALLED: send_process()\n",
             radio().id());
1227  #endif
1228       node_id_t port, port_target, destination;
1229
1230  #ifdef HWY_SEND_DEBUGG
```

```
1231        debug().debug("---------------SENDRECV--------------\n"
               );
1232        for(int i = 0; i<len; ++i)
1233             debug().debug("Data[%d]: %d\n", i, data[i]);
1234        debug().debug("---------------/SENDRECV--------------\n
               ");
1235    #endif
1236
1237        // Check if we are still in the cluster that
1238        // originated the request.
1239        if ( from == cluster().parent() || from == radio().id()
                )
1240        {
1241             // If the current node is not the port,
1242             // continue the way to the port.
1243    #ifdef ISENSE_APP
1244             port = ( data[4] << 8 ) | data[3];
1245             port_target = ( data[8] << 8 ) | data[7];
1246             destination = ( data[2] << 8 ) | data[1];
1247    #else
1248             port = ( data[8] << 24 ) | ( data[7] << 16 ) | (
                    data[6] << 8 ) | data[5];
1249             port_target = ( data[16] << 24 ) | ( data[15] <<
                    16 ) | ( data[14] << 8 ) | data[13];
1250             destination = ( data[4] << 24 ) | ( data[3] << 16
                    ) | ( data[2] << 8 ) | data[1];
1251    #endif
1252             if( port_target == radio().id() )
1253             {
1254    #ifdef HWY_DEBUG
1255                  debug().debug( "MERDA!!! %x; %x", cluster().
                         cluster_id(), destination );
1256    #endif
1257                  return;
1258             }
1259
```

```
1260            if ( port != radio().id() )
1261            {
1262                 routing_iterator rit = routing_table_.find(
                        port );
1263                 if( rit == routing_table_.end() )
1264                 {
1265 #ifdef HWY_DEBUG
1266                      debug().debug( "PITFALL3 Managed" );
1267 #endif
1268                      // As it wasn't in the routing table,
1269                      // We try to send it straight
1270                      radio().send( port, len, data );
1271                      return;
1272                 }
1273 #ifdef TRACK_SEND_MSG
1274                 debug().debug( "(%d)Process_send sending to %
                        x through %x",cluster().is_cluster_head(),
                         destination, rit->second );
1275 #endif
1276                 radio().send( rit->second, len, data );
1277            }
1278            // Send the message to the other cluster port.
1279            else
1280            {
1281                 radio().send( port_target, len, data );
1282            }
1283       }
1284       else
1285       {
1286            // Create a "2" message.
1287            *data += 1;
1288
1289            if( cluster().is_cluster_head() )
1290            {
1291 #ifdef TRACK_SEND_MSG
1292                 debug().debug( "Process_send port target and
```

```
                    leader" );
1293 #endif
1294                 cluster_head_work( from, len, data );
1295             }
1296             else
1297             {
1298 #ifdef TRACK_SEND_MSG
1299                 debug().debug( "Process_send port target,
                        sending to %x through %x", destination,
                        cluster().parent() );
1300 #endif
1301                 radio().send( cluster().parent(), len, data )
                        ;
1302             }
1303       }
1304 #ifdef HIGHWAY_METHOD_DEBUG
1305       debug().debug("@@ %x METHOD_ENDED: send_process()\n",
             radio().id());
1306 #endif
1307 }
1308
1309 // -----------------------------------------------------
1310
1311 template<typename OsModel_P,
1312          typename RoutingTable_P,
1313          typename Cluster_P,
1314          typename Neighbor_P,
1315          uint16_t MAX_CLUSTERS>
1316 void
1317 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
1318 cluster_head_work( node_id_t from, size_t len, block_data_t
       *data )
1319 {
1320       if(!enabled_)
1321            return;
```

```
1322 #ifdef HIGHWAY_METHOD_DEBUG
1323       debug().debug("@@ %x METHOD_CALLED: cluster_head_work()
           \n", radio().id() );
1324 #endif
1325       node_id_t sender;
1326
1327       get_msg_highway( &msg_highway_, data );
1328
1329       if ( msg_highway_.msg_id == CANDIDACY ) // Add the port
           candidate to the queue.
1330       {
1331            if( msg_highway_.sid_target == radio().id() )
1332                 return;
1333            // Check that the port candidate is better than
1334            // the stored candidate
1335            highway_iterator it = ports_queue_.find(
               msg_highway_.sid_target );
1336 #ifdef STABLE
1337            if( it == ports_queue_.end() || it->second.second.
               first > msg_highway_.hops )
1338 #else
1339            if( true )
1340 #endif
1341            {
1342                 ports_queue_[msg_highway_.sid_target] = entry
                    ( source_target( msg_highway_.source,
                    msg_highway_.target ), hops_ack(
                    msg_highway_.hops, 0 ));
1343 #ifdef HWY_DEBUG
1344                 debug().debug( "(%d)HWY received candidacy
                    from %x; %x; %x; %x", cluster().
                    is_cluster_head(), msg_highway_.source,
                    msg_highway_.target, msg_highway_.
                    sid_source, msg_highway_.sid_target );
1345 #endif
1346                 if( not cand_timer_set_ )
```

```
1347                {
1348                     cand_timer_set_ = true;
1349                     timer().template set_timer<self_type, &
                            self_type::candidacies_timeout>(
                            discovery_time_ , this, (void *) 0 );
1350                }
1351           }
1352      }
1353      else if ( msg_highway_.msg_id == PORT_REQ2 ) // Accept
             or reject the highway request.
1354      {
1355 #ifdef HWY_DEBUG
1356           debug().debug( "HWY received port request %x; %x;
                %x; %x", msg_highway_.source, msg_highway_.
                target, msg_highway_.sid_source, msg_highway_.
                sid_target );
1357 #endif
1358           // iff there is no highway set or the ack of the
1359           // current one are too high accept or is the
1360           // same highway
1361           highway_iterator it = highway_table_.find(
                msg_highway_.sid_source );
1362           if( it == highway_table_.end() || it->second.
                second.second > max_acks_ || ( it->second.first
                .first == msg_highway_.target && it->second.
                first.second == msg_highway_.source ) )
1363           {
1364 #ifdef HWY_DEBUG
1365                debug().debug( "HWY acking %x; %x; %x; %x",
                     msg_highway_.source, msg_highway_.target,
                     msg_highway_.sid_source, msg_highway_.
                     sid_target );
1366 #endif
1367                highway_table_[msg_highway_.sid_source] =
                     entry( source_target( msg_highway_.target,
                      msg_highway_.source ), hops_ack(
```

```
                    msg_highway_.hops, 0 ) );
1368                msg_highway_.msg_id = PORT_ACK;
1369                set_msg_highway( buffer_, msg_highway_.msg_id
                    , msg_highway_.hops, msg_highway_.source,
                    msg_highway_.target, msg_highway_.
                    sid_source, msg_highway_.sid_target );
1370                radio().send( from, HWY_MSG_SIZE, buffer_ );
1371           }
1372      }
1373      // Establish the port.
1374      else if ( msg_highway_.msg_id == PORT_ACK2 )
1375      {
1376 #ifdef CTI_VISOR
1377           debug().debug( "HWY_ADDED; %x; %x; %x; %x; %d",
                msg_highway_.source, msg_highway_.target,
                msg_highway_.sid_source, msg_highway_.
                sid_target, msg_highway_.hops );
1378 #endif
1379           highway_table_[msg_highway_.sid_target] = entry(
                source_target( msg_highway_.source,
                msg_highway_.target ), hops_ack( msg_highway_.
                hops, 0 ) );
1380      }
1381      // Remove the port or highway.
1382      else if ( msg_highway_.msg_id == PORT_NACK2 )
1383      {
1384           highway_iterator it = highway_table_.find(
                msg_highway_.sid_target );
1385           if( it != highway_table_.end() &&  it->second.
                first.first == msg_highway_.sid_source || it->
                second.first.second == msg_highway_.sid_target
                )
1386           {
1387 #ifdef CTI_VISOR
1388                debug().debug( "HWY_DEL; %x; %x; %x; %x",
                    msg_highway_.source, msg_highway_.target,
```

```
                    msg_highway_.sid_source, msg_highway_.
                    sid_target );
1389 #endif
1390                highway_table_.erase(msg_highway_.sid_target)
                    ;
1391           }
1392
1393           // Try to renegotiate
1394           if( not cand_timer_set_ )
1395                candidacies_timeout( (void *) 0 );
1396      }
1397      else
1398      {
1399           // Send the msg_ack and call the registered
1400           // (if exists) receiving method.
1401           if ( *data == SEND2 )
1402           {
1403 #ifdef ISENSE_APP
1404                sender = (data[6] << 8) | data[5];
1405 #else
1406                sender = ( data[12] << 24 ) | ( data[11] <<
                    16 ) | ( data[10] << 8 ) | data[9];
1407 #endif
1408 #ifdef HWY_SEND_DEBUG
1409                debug().debug( "---------------HEAD_WORK_RECV
                    --------------\n" );
1410                for( int i = 0; i<len; ++i )
1411                  debug().debug( "Data[%d]: %d\n", i, data
                       [i] );
1412                debug().debug( "---------------HEAD_WORK_RECV
                    --------------\n" );
1413 #endif
1414 #ifdef TRACK_SEND_MSG
1415                debug().debug( "Arrived to the leader" );
1416 #endif
1417                send( false, sender, 0, data );
```

```
1418                 if( reg_callback_ ) hwy_recv_callback_(sender
                        , len-SEND_OVERHEAD, &data[SEND_OVERHEAD])
                        ;
1419                 else debug().debug( "No one registered the
                        receive callback :O!" );
1420           }
1421           //Count the received ACKs from the highway
1422           else if ( *data == ACK2 )
1423           {
1424                 highway_iterator it = highway_table_.find(
                        sender );
1425                 if( it != highway_table_.end() && it->second.
                        second.second > -100 )
1426                   it->second.second.second -= 4;
1427           }
1428     }
1429 #ifdef HIGHWAY_METHOD_DEBUG
1430     debug().debug("@@ %x METHOD_ENDED: cluster_head_work()\
           n", radio().id() );
1431 #endif
1432 }
1433
1434 // -------------------------------------------------------
1435
1436 template<typename OsModel_P,
1437          typename RoutingTable_P,
1438          typename Cluster_P,
1439          typename Neighbor_P,
1440          uint16_t MAX_CLUSTERS>
1441 void
1442 HighwayCluster<OsModel_P, RoutingTable_P, Cluster_P,
       Neighbor_P, MAX_CLUSTERS>::
1443 send( bool send_ack, node_id_t destination, size_t len,
       block_data_t *data )
1444 {
1445     if(!enabled_)
```

```
1446            return;
1447  #ifdef HIGHWAY_METHOD_DEBUG
1448        debug().debug( "@@ %x METHOD CALLED: send()\n", radio()
              .id() );
1449  #endif
1450        //Check if the highway is still valid.
1451
1452        highway_iterator it = highway_table_.find( destination
              );
1453        if( it == highway_table_.end() || it->second.second.
              second > max_acks_ )
1454        {
1455             highway_table_.erase(destination);
1456             return;
1457        }
1458
1459        node_id_t port = it->second.first.first;
1460        node_id_t port_target = it->second.first.second;
1461  #ifdef TRACK_SEND_MSG
1462        debug().debug( "TRACK: sending to %x through %x",
              destination, port );
1463  #endif
1464        if(send_ack )
1465        {
1466            buffer_[0] = SEND;
1467            if( it->second.second.second < 100 )
1468                 it->second.second.second += 3;
1469        }
1470        else
1471        {
1472            buffer_[0] = ACK;
1473        }
1474  #ifdef ISENSE_APP
1475        buffer_[1] = destination & 0xFF;
1476        buffer_[2] = ( destination >> 8 ) & 0xFF;
1477        buffer_[3] = port & 0xFF;
```

```
1478        buffer_[4] = ( port >> 8 ) & 0xFF;
1479        buffer_[5] = radio().id() & 0xFF;
1480        buffer_[6] = ( radio().id() >> 8 ) & 0xFF;
1481        buffer_[7] = port_target & 0xFF;
1482        buffer_[8] = ( port_target >> 8 ) & 0xFF;
1483    #else
1484        buffer_[1] = destination & 0xFF;
1485        buffer_[2] = ( destination >> 8 ) & 0xFF;
1486        buffer_[3] = ( destination >> 16 ) & 0xFF;
1487        buffer_[4] = ( destination >> 24 ) & 0xFF;
1488        buffer_[5] = port & 0xFF;
1489        buffer_[6] = ( port >> 8 ) & 0xFF;
1490        buffer_[7] = ( port >> 16 ) & 0xFF;
1491        buffer_[8] = ( port >> 24 ) & 0xFF;
1492        buffer_[9] = radio().id() & 0xFF;
1493        buffer_[10] = ( radio().id() >> 8 ) & 0xFF;
1494        buffer_[11] = ( radio().id() >> 16 ) & 0xFF;
1495        buffer_[12] = ( radio().id() >> 24 ) & 0xFF;
1496        buffer_[13] = port_target & 0xFF;
1497        buffer_[14] = ( port_target >> 8 ) & 0xFF;
1498        buffer_[15] = ( port_target >> 16 ) & 0xFF;
1499        buffer_[16] = ( port_target >> 24 ) & 0xFF;
1500    #endif
1501
1502    #ifdef HWY_SEND_DEBUG
1503        debug().debug( "---------------ENCAPSULATING
              ----------------\n" );
1504    #endif
1505
1506        for (int i = 0; i < len; ++i)
1507        {
1508            buffer_[i+SEND_OVERHEAD] = data[i];
1509    #ifdef HWY_SEND_DEBUG
1510            debug().debug( "Data item %d: %d\n", i, data[i] );
1511    #endif
1512        }
```

```
1513
1514
1515 #ifdef HWY_SEND_DEBUG
1516       for ( int i = 0; i < len+SEND_OVERHEAD; ++i )
1517       {
1518             debug().debug( "Buffer item %d: %d\n", i, buffer_[
                     i] );
1519       }
1520       debug().debug( "---------------/ENCAPSULATING
                  ----------------\n" );
1521 #endif
1522
1523       process_send( radio().id(), len+SEND_OVERHEAD, buffer_
                  );
1524 #ifdef HIGHWAY_METHOD_DEBUG
1525       debug().debug( "@@ %x METHOD ENDED: send()\n", radio().
                  id() );
1526 #endif
1527 }
1528
1529 }
1530 #endif
```

Bibliography

[1] FRONTS Project. **Foundations of Adaptive Networked Societies of Tiny Artefacts**. http://fronts.cti.gr. 1

[2] WISEBED Project. **Wireless Sensor Network Testbeds**. http://www.wisebed.eu. 1

[3] T. Baumgartner, I. Chatzigiannakis, S. Fekete, C. Koninis, A. Kröller, and A. Pyrgelis. **Wiselib: A Generic Algorithm Library for Heterogeneous Sensor Networks**. In *7th European Conference on Wireless Sensor Networks (EWSN)*, **5970** of *Lecture Notes in Computer Science*, pages 162–177. Springer, 2010. 7

[4] S. Fischer S. Fekete, A. Kröller and D. Pfisterer. **Shawn: The fast, highly customizable sensor network simulator**. In *4th Intl. Conference on Networked Sensing Systems (INSS)*, page 299, 2007. 7

[5] WISEBED. **Wiselib webpage**. https://www.wiselib.org. 13

[6] TUBS and UZL. **SHAWN wiki main page**. https://www.itm.uni-luebeck.de/ShawnWiki/index.php/Main_Page. 13, 32

[7] Coalesenses GmbH. **iSense Wireless Sensor Network Hardware Modules**. http://www.coalesenses.com/index.php?page=isense-hardware. 13

[8] WISEBED and FRONTS et al. **Wiselib subversion repository**. https://svn.itm.uni-luebeck.de/wisebed/wiselib/trunk/. 14

[9] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000. 14

[10] BOOST Project. **BOOST C++ libraries**. http://www.boost.org. 14

[11] CGAL Project. **CGAL: Computational Geometry Algorithms Library**. http://www.cgal.org. 14

[12] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000. 18

[13] 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. 30

[14] Hugo Hernández, Tobias Baumgartner, Maria J. Blesa, Christian Blum, Alexander Kröller, and Sándor P. Fekete. **A Protocol for Self-Synchronized Duty-Cycling in Sensor Networks: Generic Implementation in Wiselib**. *CoRR*, **abs/1010.4385**, 2010. 30

BIBLIOGRAPHY

[15] Ioannis Chatzigiannakis, Christos Koninis, Panagiota Panagopoulou, and Paul Spirakis. **Distributed Game-Theoretic Vertex Coloring**. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems*, **6490** of *Lecture Notes in Computer Science*, pages 103–118. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-17653-1_9.30

[16] Tobias Baumgartner, Daniel Bimschas, Sándor Fekete, Stefan Fischer, Alexander Kröller, Max Pagel, and Dennis Pfisterer. **Demo Abstract: Bridging the Gap between Simulated Sensor Nodes and the Real World**. In Pedro Marron, Thiemo Voigt, Peter Corke, and Luca Mottola, editors, *Real-World Wireless Sensor Networks*, **6511** of *Lecture Notes in Computer Science*, pages 174–177. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-17520-$6_1$9.30

[17] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, 2004. 31

[18] Sándor P. Fekete, Alexander Kröller, Dennis Pfisterer, and Stefan Fischer. **Algorithmic Aspects of Large Sensor Networks**. In *Proceedings of Mobility and Scalability in Wireless Sensor Networks (MSWSN' 06)*, pages 141–152. CTI Press, 2006. 36

[19] Alexander Kröller, Dennis Pfisterer, Carsten Buschmann, Sándor P. Fekete, and Stefan Fischer. **Shawn: A new approach to simulating wireless sensor networks**. In *Proceedings of the Design, Analysis, and Simulation of Distributed Systems Symposium 2005 (DASD' 05)*, pages 117–124, April 2005. 36

[20] NXP Semiconductors. **Jennic Wireless Microcontrollers JN5139**. `http://www.jennic.com/products/wireless_microcontrollers/jn5139`. 37

[21] RXTX Project. **RXTX Wiki**. `http://rxtx.qbang.org/wiki/index.php/Main_Page`. 42

[22] FRONTS Project. **FRONTS Project public documents**. `http://fronts.cti.gr/index.php/deliverables`. 71

[23] Orestis Akribopoulos, Lorenzo Bergamini, Andreas Cord-Landwehr, Christos Koninis, and Antoni Segura Puimedon. *Distributed Self-organized Societies of Tiny Artefacts: Design & Implementation*. Lulu Publishers, Raleigh, N.C., 2011. 71

[24] S Katti, H Rahul, Wenjun Hu, Dina Katabi, and J Crowcroft. **XORs in the Air: Practical Wireless Network Coding**. *IEEE/ACM Transactions on Networking*, **16**(3):497–510, 2008.

[25] Shlomi Dolev and Nir Tzachar. **Empire of colonies: Self-stabilizing and self-organizing distributed algorithm**. *Theor. Comput. Sci.*, **410**:514–532, February 2009.

[26] 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 (SENSAPPEAL)*, **29** of *Lecture Notes of the Institute for Computer Sciences, Social-Inf*, pages 68–87. Springer, 2009.

[27] Tobias Baumgartner, Ioannis Chatzigiannakis, Maick Danckwardt, Christos Koninis, Alexander Kröller, Georgios Mylonas, Dennis Pfisterer, and Barry Porter. **Virtualising Testbeds to Support Large-Scale Reconfigurable Experimental Facilities**. In Jorge Silva, Bhaskar Krishnamachari, and Fernando Boavida, editors, *Wireless Sensor Networks*, **5970** of *Lecture Notes in Computer Science*, pages 210–223. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-11917-$0_1$4.

[28] O. Akribopoulos, I. Chatzigiannakis, C. Koninis, and E. Theodoridis. **A web services-oriented architecture for integrating small programmable objects in the web of things**. In *3rd International Conference on Developments in eSystems Engineering (DeSE 2010)*, pages 70–75, 2010.