Highway Construction for Wireless Sensor Networks*

Antoni Segura Maria J. Blesa

Dept. Llenguatges i Sistemes Informàtics
Universitat Politècnica de Catalunya
E-08034 Barcelona, Spain

*A. Segura is partially supported by the FET Programme (FET Proactive Initiative: Pervasive Adaptation) under contract numbers 215270 within the EU FP7-ICT-2007-1 (FRONTS). M. Blesa is partially supported by the New Paradigms and Experimental Facilities of the Small/medium-scale focused research projects (STREPs) under contract number FP7-ICT-2007-2 ICT-2007.1.6 (WISEBED).

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.

Contents

1

Introduction

The work that describes this document is developed under the support and scope of two European projects from the 6th 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

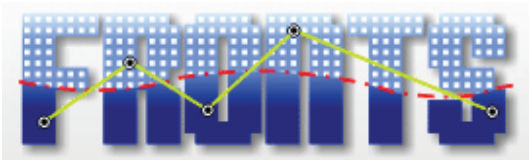


Figure 1.1: The FRONTS project logo

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 this 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.

1. INTRODUCTION

In order to study the possibilities and nature of this 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*.

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 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.

Eleven partners conform the FRONTS project. Details on the participating sites and their coordinators can be found on the table 1.1, being the Computer Technology Institute (CTI), in Greece, the project coordinator.

1.2 Highway project

The Highway project is the research for a distributed algorithm to find communication paths on hierarchical Wireless Sensor Network topologies. It 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 1.2. 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's Wiselib project, and the second part has been determined by the usage of the tools, scripts and occasional administration of the available testbeds[1] 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 **??**.

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

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

	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

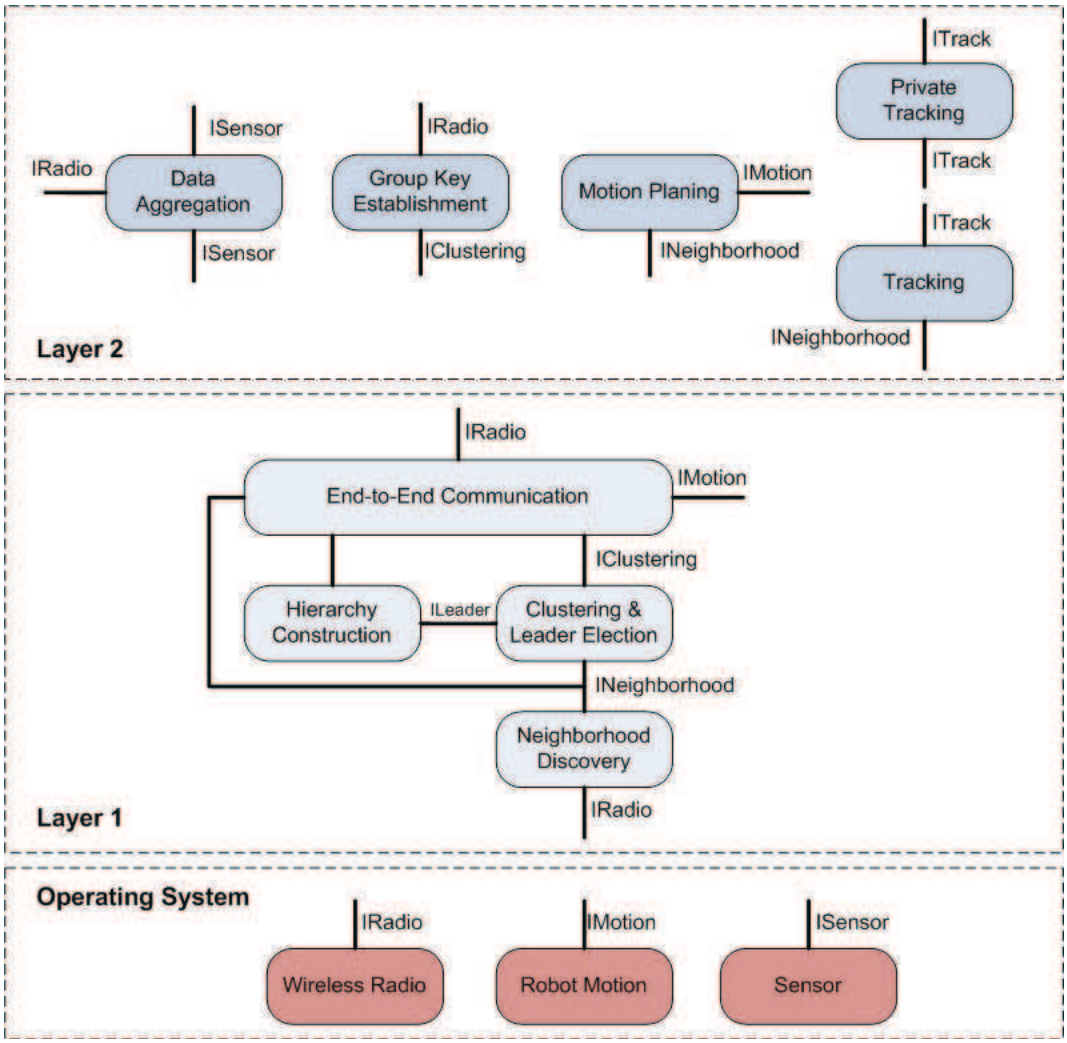


Figure 1.2: The FRONTS software architecture.

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 1.2 were altered to factor out a common piece of functionality that finally became an algorithm of its own[1] for all the others to use. The factoring out proved to be effective as code size is concerned. However, done as it was a posteriori (from the initial implementations of the algorithms), it forced a compromise that had most of the involved parties change the inner workings of their algorithms.

The networking layer structure, as seen in figure 1.2 is built on the base of the operating system (which is accessed through the Wiselib, as seen in page 17). 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 to classify the nodes in groups with one leader each. The result of applying the clustering algorithm, is that, not only all the nodes become associated with a higher order entity called cluster(it is common for a cluster to be formed by just one node, and even in this cases it is considered a higher entity, cause it can annex free surrounding nodes), 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, which are basically the information of a change in cluster association and paths to the leader, there are two modules, the *Highway module* and the *End to End communication module*. The former centers its algorithm around

[1] the algorithm in question is the neighborhood discovery module. In retrospect, it's quite evident that this is a part that should have stood up on its own from the beginning, but due to the fact that FRONTS started with a more theoretical base than the one it ended up having, every one of the participating sites had already devised their own neighborhood discovering methods.

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 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 algorithm 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 testbed. For a lengthier description of the testbeds and their operation, look up page 37.

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(3) project, which is the library of algorithms and data structures developed in the scope of the WISEBED project. Using the Wiselib enabled 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(4) simulator and the iSense(5) sensor hardware.

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

2.1 Wiselib

The Wiselib is an algorithm library 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(6), 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++(7), and makes a widespread use of templates in the same fashion as Boost and CGAL 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

The way in which code reutilization and modularity is regulated in Wiselib 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 figure 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*. 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;
```

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

```
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 iSense, 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.

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

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.

[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 (8, 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.

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 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 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;
```

[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.

```
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

[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.

a sample call[1] 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 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

[1] 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.

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.

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

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

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  }
```

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

[1]For a good description an generalization of this issue with extensive data to back it (although coming not just from embedded background, read (9, p. 28-33).

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[1]. 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.

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[2], 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

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

[2]Further information about Shawn can be found at SHAWN's wiki(4)

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 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'
14 Simulation: Task done 'random_seed'
15 Simulation: Running task 'prepare_world'
16 DiskGraphModel: Transmission range set to [300]
17 Simulation: Task done 'prepare_world'
18 Simulation: Running task 'chain_transm_model'
19 Simulation: Task done 'chain_transm_model'
20 Simulation: Running task 'rect_world'
21 Simulation: Task done 'rect_world'
22 Simulation: Running task 'simulation'
23 ------------------------------ BEGIN ITERATION 0
24 SEND JOIN [11 , 0 , 0]
25 SEND JOIN [11 , 10 , 0]
26 SEND JOIN [11 , 20 , 0]
27 ...
28 ------------------------------ DONE ITERATION 0
29   [ 100 active, 0 sleeping, 0 inactive ]
30
31 ------------------------------ BEGIN ITERATION 1
32 ...
33
34 ------------------------------ DONE ITERATION 1500
35   [ 100 active, 0 sleeping, 0 inactive ]
36
37 Simulation: Task done 'simulation'
```

```
38 Simulation: Running task 'dump_transmission_stats'
39 ---- stats_chain transmission model information
       -----------------
40 general all_types messages 151203
41 general all_types size 14544003
42 general N7wiselib14WiselibMessageIihlEE messages 151203
43 general N7wiselib14WiselibMessageIihlEE size 14544003
44 ----------------------------------------------------------------

45 Simulation: Task done 'dump_transmission_stats'
46 Simulation: Running task 'connectivity'
47 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.

2.3 iSense

[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

Figure 2.1: The core module of an iSense sensor

As stated in the Highway project section, more precisely on the page 10, 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 *Coalesenses* product for Wireless Sensor Networks called iSense.

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 2.1 with UART connectible ports that enable the attachment of additional boards (seen in figure 2.2) 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 2.2, it is composed of a 32-bit RISC jennic microcrontroller(10) 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

[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

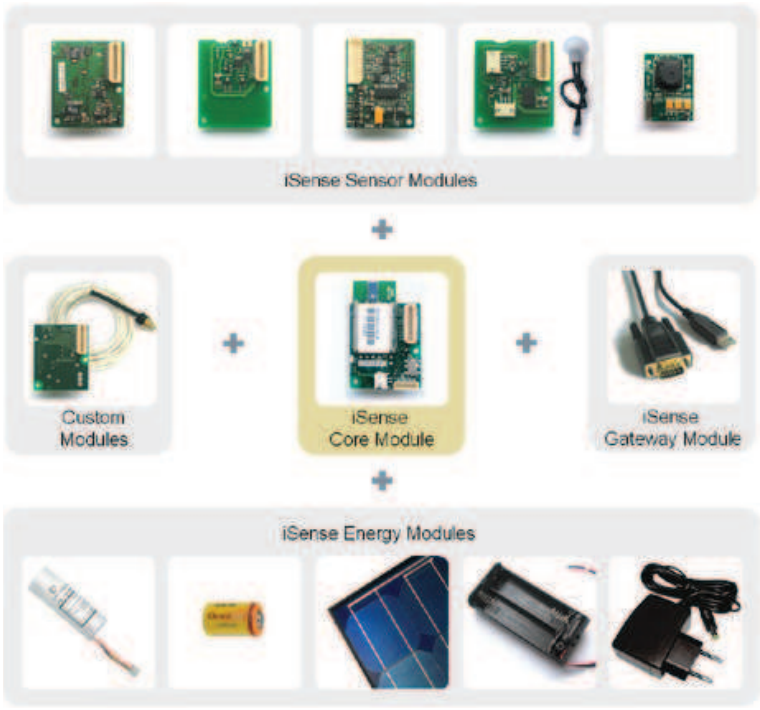


Figure 2.2: iSense core and modules architecture

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.

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 project 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

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

sensors.

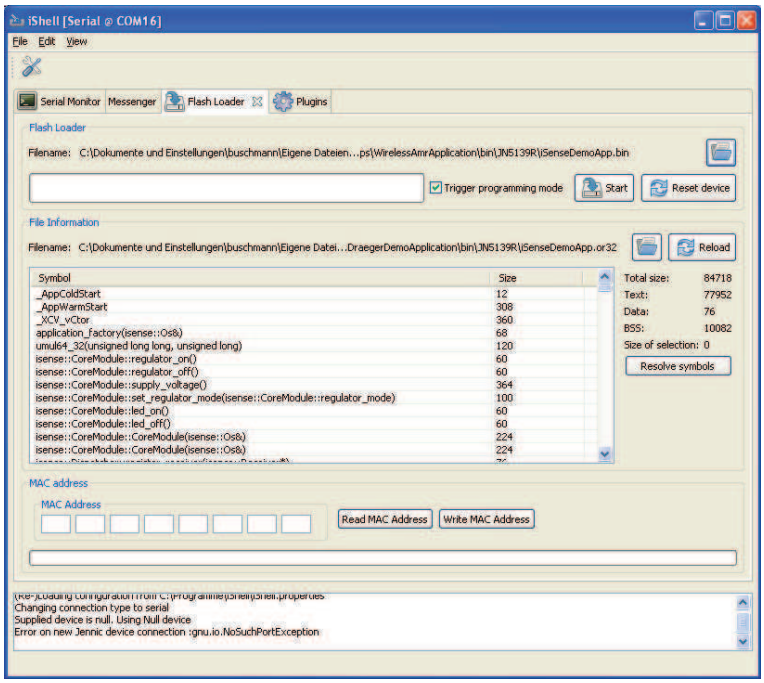


Figure 2.3: iShell application for loading and running iSense applications

The development[1] work flow of the iSense ecosystem, not considering the testbeds 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.

[1]The development environment tools work both in Microsoft Windows and Linux platforms and are available at http://www.coalesenses.com/index.php?page=development-environent

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 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

[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

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 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(11) 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 Highway algorithm consists in devising 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. 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 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 among 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 this 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.

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.

Whereas 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.

The nodes in a cluster that are not elected as leaders by the clustering algorithm are, from the perspective of the Highway algorithm, mainly deemed as potentially communicating nodes that notify about other clusters and, if chosen to form part of a highway, pass the highway level messages around. Leaders, on top of the communicating capabilities, are entitled to initiate negotiations with other leaders to establish highways between them, disable and/or change them.

The communication capabilities of the cluster leaders are basically being able to encapsulate data which will arrive to the destination cluster leader without any intervention by the highway client (typically the end to end module). 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.

4

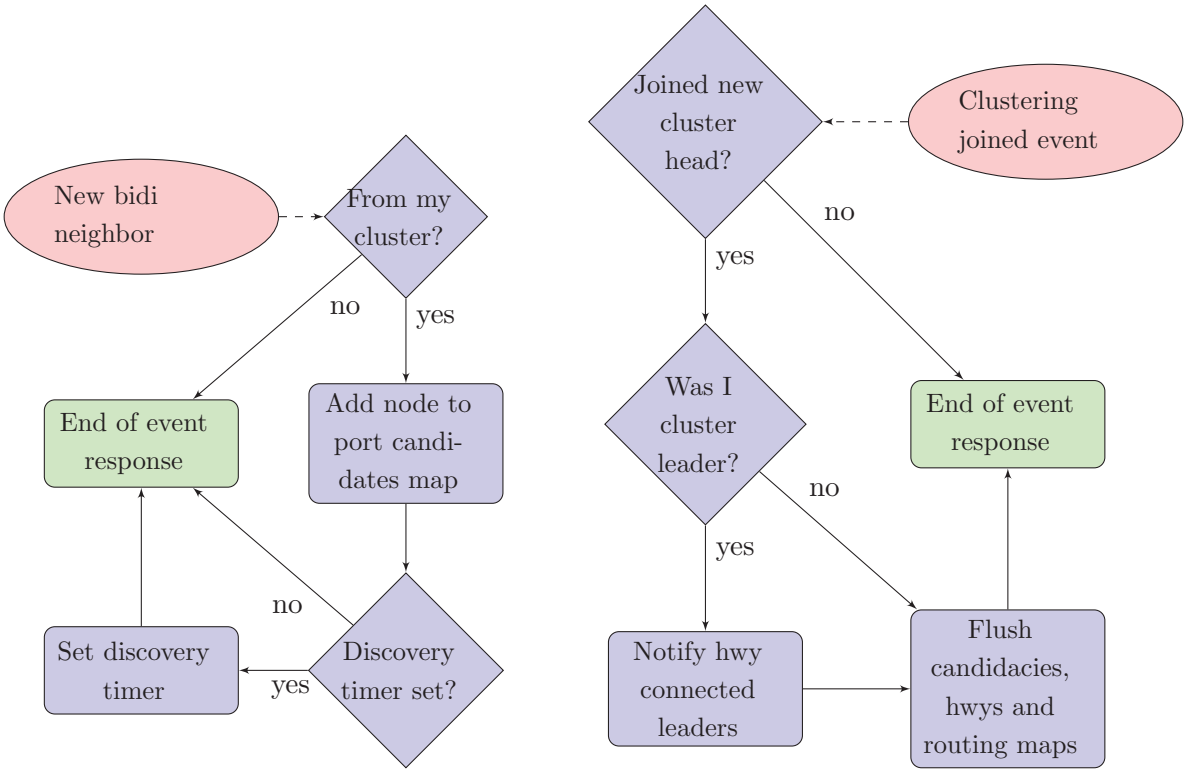
Implementation

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.

4.1 One to one based highways

The first diagrams, 4.1a and 4.1b 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.

As a result of the previous flowcharts, granted that there are multiple nodes and clusters, the 4.1a 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 4.2a, 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



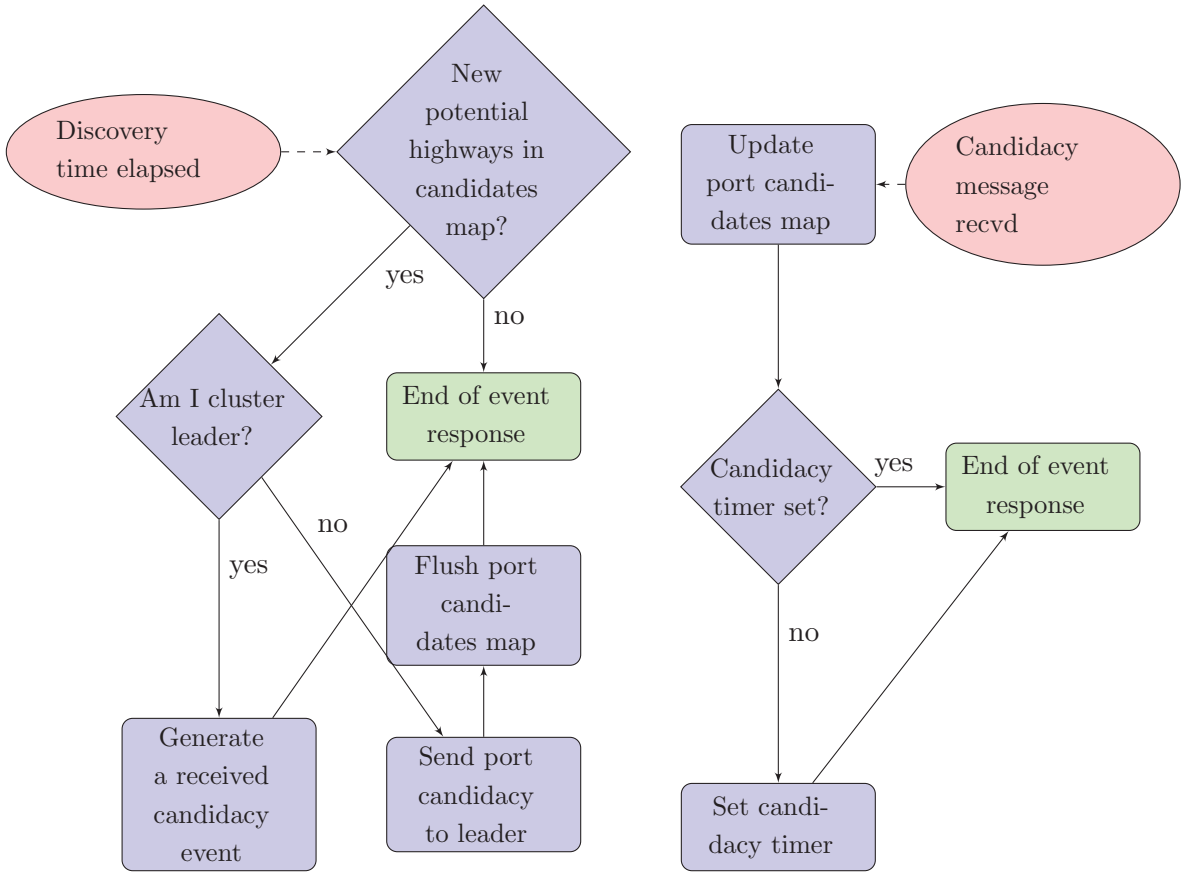
(a) Highway algorithm response to a neighborhood discovery new bidirectional neighbor event

(b) Highway algorithm response to a clustering joined event

Figure 4.1: Algorithm diagram of the response to the clustering and neighborhood discovery related events.

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 4.1a diagram, sets a message receiving window, as displayed in the 4.2b figure.

Once the candidacies timer elapses its alloted time, it generates an event that is processed in the way depicted in figure 4.3a. As the reader can see, it basically checks if any of the highway candidates present real opportunities to create working highways 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



(a) Highway algorithm response to a discovery timer event

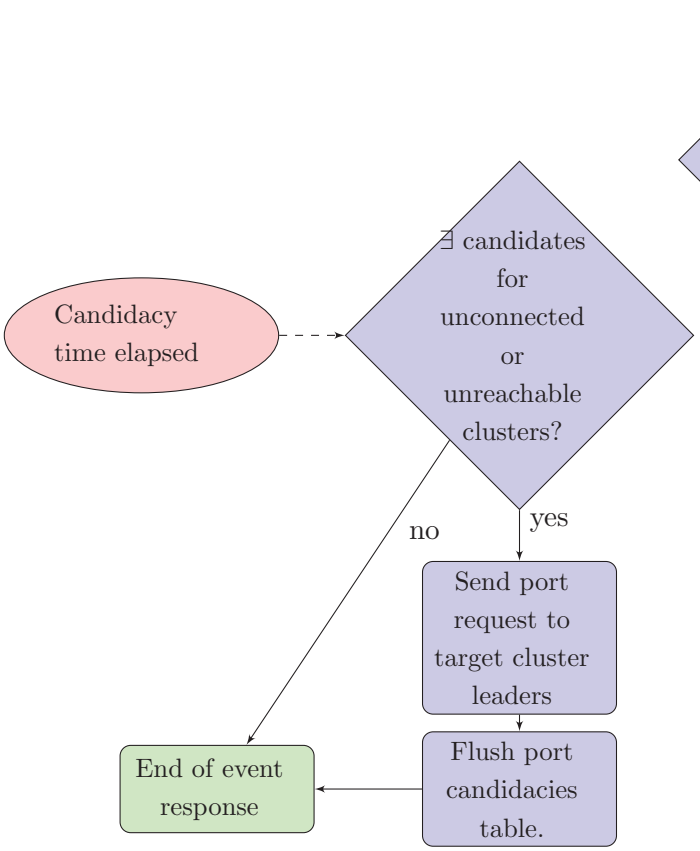
(b) Highway algorithm response to a candidacy request

Figure 4.2: Highway algorithm response to discovery and candidacy message events.

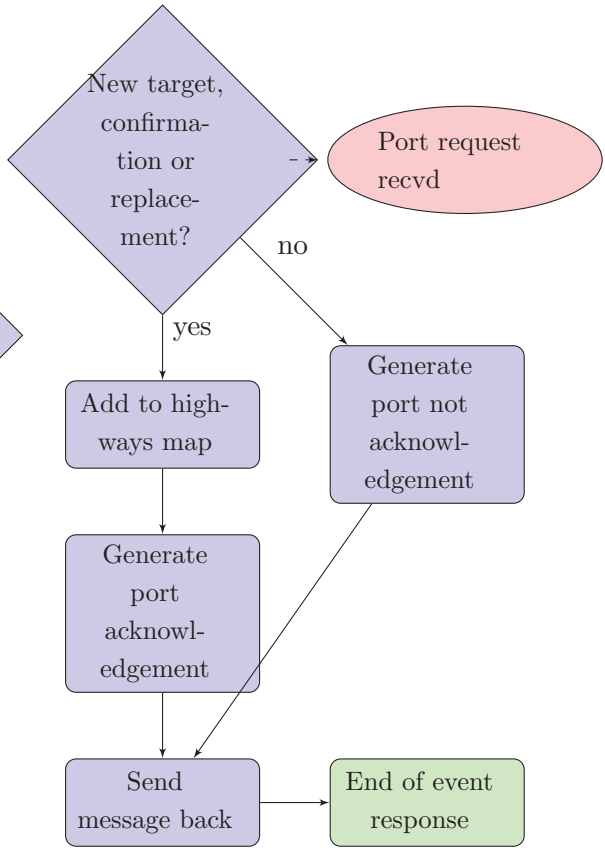
the solution, allows the devices to operate without any hindrance and, in any case, doesn't bar from treating a response if it is received.

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 4.3b. 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).

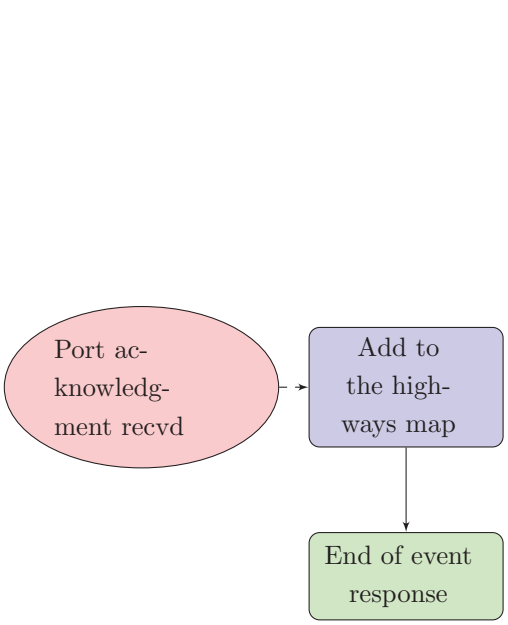
When a node receives an acknowledgment (figure 4.3c), it automatically places the acknowledged highway into its working highways map. In contrast, when a non-acknowledgment is received (figure 4.3d), it is removed from the highways map and the process of the figure 4.3a is invoked without the usual candidacy 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 4.1b, it tries to notify the receiving ends of its now void highways of the invalidity of the paths.



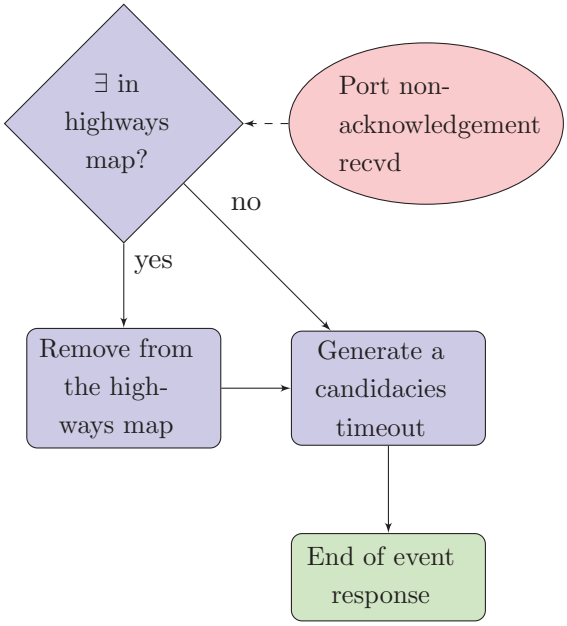
(a) Highway algorithm response to a candidacy timer event on a leader



(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 4.3: Highway algorithm port negotiation.

5

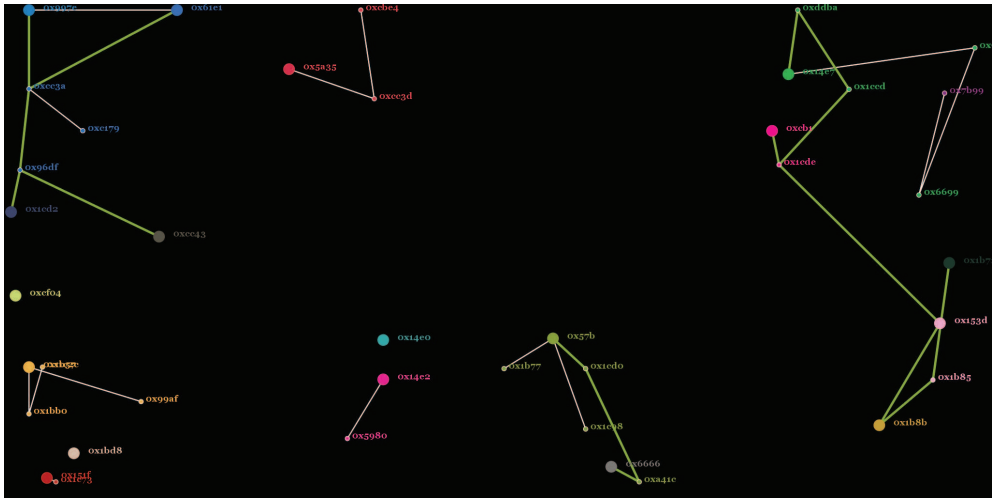
Experiments

5.1 Without node failures

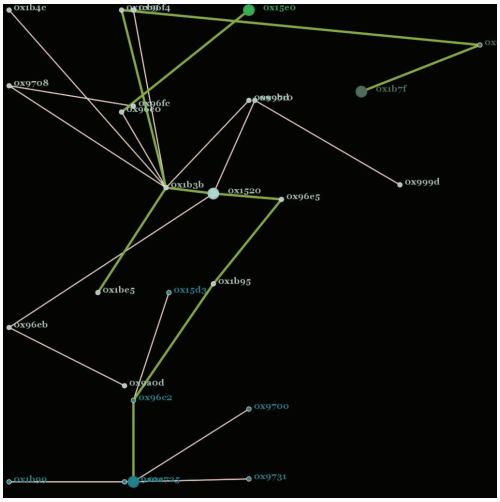
The Highway module sits directly above of the clustering module, and thus runs also on top of the neighborhood discovery module. Thus, the outcome to be obtained from the experiments is tightly coupled to those from the clustering and neighborhood discovery modules described above. In Figure 5.1 we include 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 5.1a)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



(a) Disconnected network **(b)** Di



(c) Connected network

Figure 5.1: 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.

longevity was found to be short enough so that no high amount of graph connectivity was attained.

The snapshot of the CTI testbed (see Figure 5.1b), 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 beliefs

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.

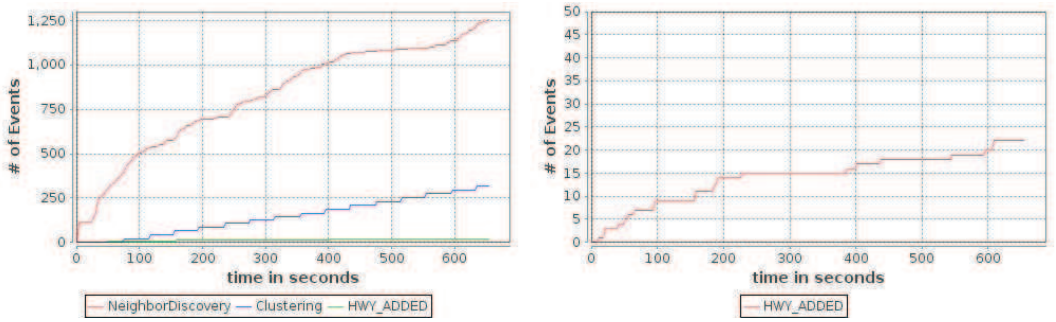
Figure **??** shows a snapshot of the UNIGE experiment at an arbitrary point in time after the highways are formed. During the experimentation in that testbed, the degree of connectivity proved itself so big, so that the output messages of highway requests had to be disabled to attain a reasonably small 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 5.2a). 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 this periods the nodes kept track of which were the extra-cluster nodes with a more direct



(a) Aggregated number of events reported by each module,

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

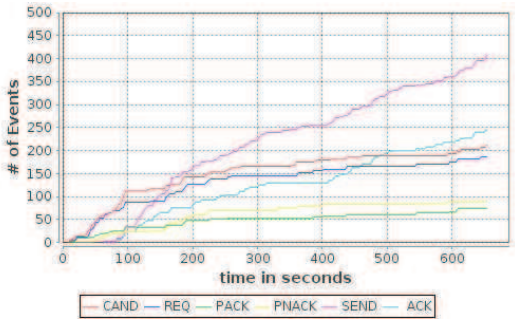
Figure 5.2: Aggregated events experiments ran in the FRONTS testbed.

path to each of the neighboring cluster leaders. However, this approach showed itself 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.

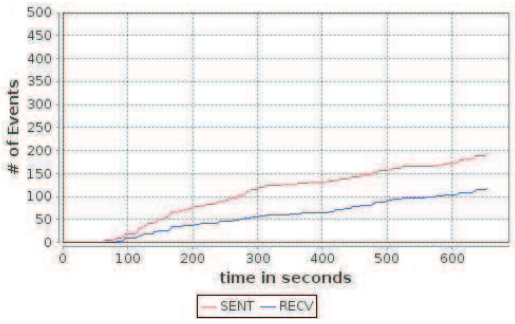
The second experiment tries to factor the energy and medium usage of the Highway module as well as its effectiveness in delivering messages. In this experiment, first we generated highway traffic by waiting $40sec$ initially, to allow the neighborhood discovery and clustering modules to stabilize, and 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 5.3a). 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 5.3: Aggregated messages experiments ran in the FRONTS testbed.

5.2 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 5.4, 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

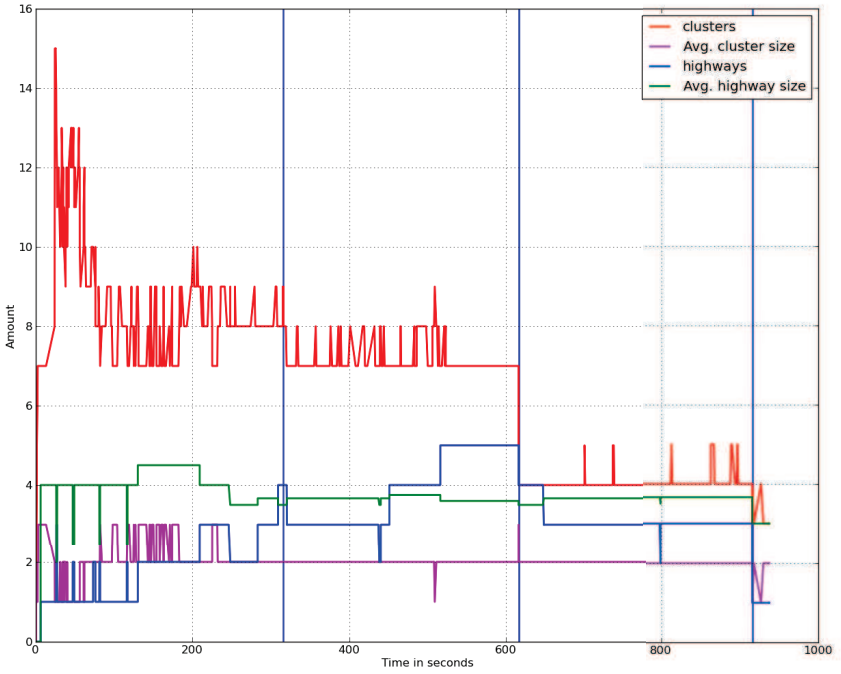


Figure 5.4: Evolution of the cluster amount, cluster average sizes, highway amount and highway average sizes, with the disconnection moments (i.e., 20% nodes failures) marked with vertical bars.

it is appreciated a bigger drop in clusters than in highways, due to a combination of the 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 5.5), 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.

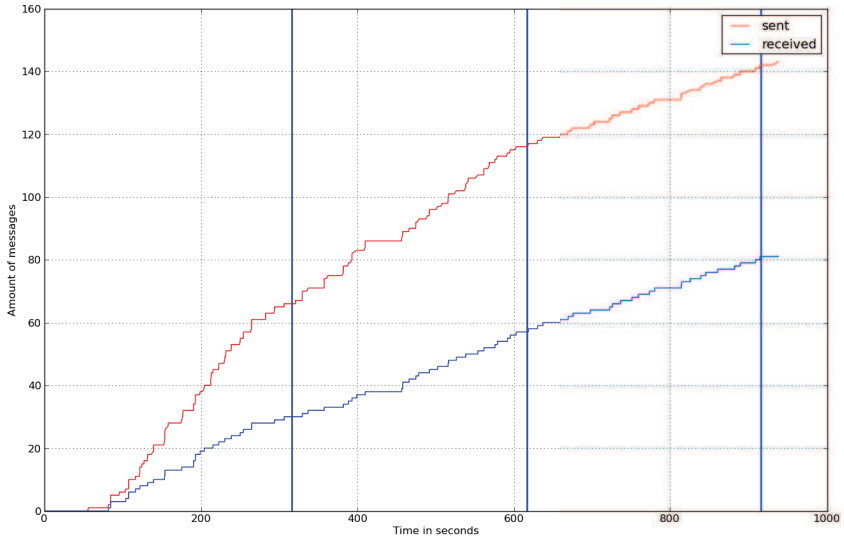


Figure 5.5: Aggregated number of messages sent and received through highways with progressively failing nodes. Disconnection moments (i.e., 20% nodes failures) are marked with vertical bars.

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] WISEBED. **Wiselib webpage**. https://www.wiselib.org. 11

[4] TUBS and UZL. **SHAWN wiki main page**. https://www.itm.uni-luebeck.de/ShawnWiki/index.php/Main_Page. 11, 29

[5] Coalesenses GmbH. **iSense Wireless Sensor Network Hardware Modules**. http://www.coalesenses.com/index.php?page=isense-hardware. 11

[6] WISEBED and FRONTS et al. **Wiselib subversion repository**. https://svn.itm.uni-luebeck.de/wisebed/wiselib/trunk/. 12

[7] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000. 12

[8] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000. 16

[9] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, 2004. 28

[10] NXP Semiconductors. **Jennic Wireless Microcontrollers JN5139**. http://www.jennic.com/products/wireless_microcontrollers/jn5139. 34

[11] RXTX Project. **RXTX Wiki**. http://rxtx.qbang.org/wiki/index.php/Main_Page. 39

[12] 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.

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