Optimization of a Self-Stabilizing Role Assignment Algorithm for Actuator/Sensor Networks

Vincenzo Curatola

Universitá degli studi di Pisa

Tesi di laurea specialistica

Maggio 2007

Universitá degli Studi di Pisa
Facoltá di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Optimization of a Self-Stabilizing Role Assignment Algorithm for Actuator/Sensor Networks

Tesi di

Vincenzo Curatola

Relatori:

Prof. Marco Avvenuti
Prof. Luigi Rizzo
Prof. Dr.-Ing. Hans Ulrich Heiss

Candidato:

Vincenzo Curatola

... é lo stupore che prende le cose,
come dopo l'amore

A. Bertolucci

Questo progetto é nato in una tiepida primavera pisana durante i mattinieri appuntamenti delle Colazioni. *Ha conosciuto uno dei piú gelidi inverni berlinesi per poi posarsi, lieve, sulla tortuosa strada delle parole in una calda e familiare mansarda pisana. Tra le righe di questo lavoro si celano emozioni, paure e speranze di questo lunghissimo anno, e con esse i tanti amici con cui ho avuto la fortuna di condividerle.*
Un sentitissimo grazie a Stefano, compagno di viaggio, esempio di come si possa diventare adulti senza mai smettere di sognare.
Grazie a Giulia, per come sa guardare oltre, per come sa poi trovare la dolcezza per arrivarci.
Grazie ad Alice ed Emilie: quando la vita mi sorride é alla loro gioia sincera che ritorno.
Grazie a Rob, spalla di tante avventure, perennemente innamorato della vita.
Grazie a Ciccio, prezioso amico. Senza clamori l'ho sempre ritrovato sui tutti i miei passi.
Grazie a Bresci, perché le imprese non riescono se non c'é qualcuno con cui poterle celebrare.
Grazie ad Alessio e Giammatteo, fraterni amici. A partire dalla Settimana del buongusto *e tornando indietro negli anni hanno sempre saputo condividere serenitá e allegra familiaritá.*
Grazie a Tony, con cui ho condiviso tutti i momenti di questi anni universitari. Tra promozioni sudate e boccioni esilaranti ho imparato molto dalla sua genuina sinceritá e coerenza.
Grazie a Nicola, capace di raro affetto e solare ironia.
Grazie a Germana, é nella luce dei suoi occhi che spesso ho ritrovato la strada.
Grazie a Petro, forza vitale e lucida leggerezza. Quando il cercare motivazioni per se stesso si trasforma in un prezioso stimolo per chi ti sta intorno.
Grazie a Raffo, per la sua incontenibile curiositá, per la semplicitá e il gusto con i quali la soddisfa.
Grazie ad Angelo, iniezione costante di gioia e voglia di vivere.
Grazie a Stefano, Giacomo, Kristina, Fabio e Fabrizio, amici e coinquilini in quella piazza Guerrazzi 2 che rimarrá per me un carissimo luogo dell'anima.
Grazie al prof. Avvenuti e al prof. Heiss, per aver creduto a questo progetto.
Grazie ad Helge, per la guida precisa e attenta di questo lavoro.
Grazie a mia sorella Irma, esempio di come si possano sposare estrema forza di volontá e grande dolcezza. Grazie a mio fratello Eugenio, speranza gioia e amicizia. Grazie ad i miei genitori, per come mi hanno aiutato a coltivare i miei sogni, lasciandomi andare quando lo chiedevo, preparandomi nel frattempo sempre buoni motivi per tornare.
Piú lunga sará la strada e piú grande sará la voglia di fare a voi ritorno.

Contents

Chapter 1

Introduction

Actuator and sensor networks (**AS-Nets**) are going to play a crucial role in changing the way to interact with our living sphere. Basically, AS-Nets consist of embedded controllers and sensor boards, working jointly with other computing devices, either mobile ones (like PDAs and Smart-Phones) or fixed ones (PCs). Applications running on a AS-Net are pervasive with the scenario in which they work. Their behavior depends on determined environmental parameters, figured out by sensors, and through actuators they can enable actions to change part of the environment itself.

Applicative range of AS-Nets is strictly related to the number of parameters analyzed, as well as to the types of actions possible to perform. An higher grade of integration of such technologies with the surrounding environment determines even wider applicative scenarios. Taking as example our own homes, technology is entering in all the components. Fridges, washing machines, ovens are already furnished with sensors and computing capabilities, some of them are even able to connect to the Internet. Objects around us are becoming even more "smart", and the possibility to let them communicate with our computing devices, whether mobile or not, opens new pervasive interactions with our surrounding sphere.

1.1 A Scenario: Smart Home

The *Smart Home* consists in a global integration among the electronic technology and household tasks. Its aim is to improve the quality of life by making easier the user interaction with domestic environments.

To make our home "*smart*", it has to be able to analyze the reality, and to autonomously trigger proper actions. Once an action is automatized it can be easily tuned on the user's habits and on his way of living. If we think of our daily tasks there is a long series that we do repeatedly, and always with the same

logic. Watering the plants, feeding of domestic animals, the most part of the food shopping, as well as the heating schedule are examples of tasks that can be automatized and easily customized. The smart home analyzes the reality through sensors disseminated in the house, then figures out the state of all the components under its control and, based on applications working on the system, triggers proper actions through actuators.

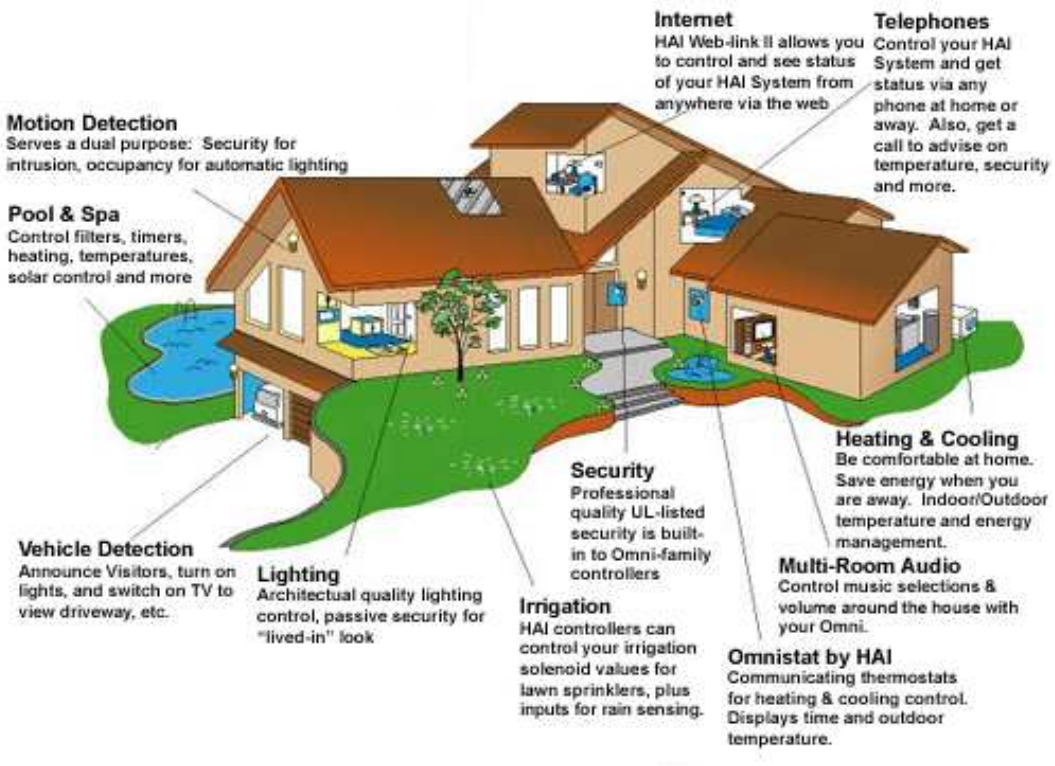

Figure 1.1: Smart Home Project

The possibility to control household tasks makes the system able to plan **energy saving** features in the daily heating schedule, by monitoring the usual family's habits. Moreover, electrical appliances usage may automatically coordinated (e.g. washing machine, dishwasher) in order to shape the energy consumption, allowing the user to stipulate more convenient contracts with the electrical company. Through position detection besides to support the alarm system, it is possible to make automatic the light turning on (off), depending on the user's presence (not presence) in a particular room. Furthermore, if the house is equipped with an integrated audio system, it would be possible to select an audio source (PC in the study, or HI-FI in the living room), and to redirect the audio stream right in the

room in which the user stays.

Food Management can be realized by equipping the kitchen by appropriate sensor boards. The fridge, the freezer and larders are in fact provided with sensors able to detect the presence of particular foods, and to estimate their expiry date as well. Moreover, such sensors are linked to a network interface, enabling them to communicate each other. By querying them, a food management application is able to say if a product is missing, or still present but getting too old. Afterwards, for instance through the pc in the study that is connected to internet, the application could send an email to the shopkeeper asking for the missing products, with a quantity proportioned to the computed family's usage. When this would not be possible, maybe because it is sunday or tuesday in the afternoon (italian shopkeeper's closing turn), a sms could be send to the mobile of a family member which is not at home yet, through the home phone.

Smart houses give answers to problems belonging to **social assistance** as well. In the health care for elderly people and in general for people with health problems, several projects are in researchers' consideration. CodeBlue [MFJWM05] is a project dealing with the continuous monitoring of patient's vital signs (e.g. blood pressure), by the use of small wearable and in-body wireless sensors. In this way the control on the disease signs is almost total, even if the intrusiveness in the person's life is considerably contained. CodeBlue, and related approaches, find in smart house a natural complement, actually extending its potentialities.

AS-Nets fit very well in such a scenario, light and position sensors support the most part of the smart house features, and actuators give the ability to the system to perform needed actions. Furthermore, AS-Nets are exploited by more powerful systems (like web applications), creating a unique coordinated system which is easily available for the user by his mobile phone, or by web.

1.2 Motivations and Goals

Smart homes show how AS-Nets can be used to improve our quality of life. Our surrounding adapts itself to our habits and changing the approach to our home. The goal of this thesis is to propose a platform over which developers can build even more impressing and user-friendly applications, without needing to deal themselves with the several design problems that a scenario like smart home presents. Which design problems? First, nodes need to communicate. In AS-Nets, and in particular in smart home, nodes are mainly wireless, and then battery charged. Wireless sensor boards have a bounded lifetime therefore often someone "dies". Moreover, not all the devices are dedicated for the smart home system. Actually, mobile phones, PDAs could be present or not, and when present they can change

positions in the scenario. The communication protocol has to care about such aspects. And when a device that was carrying out a particular task (e.g. the phone through which sms are sent) is no more present in the network, say because the user has taken it away, how the system can realize it and which node will substitute it? Furthermore, who will configure the new one? The system needs to be reconfigured, and given that such eventuality is intrinsic in this network type, such problem cannot be solved relying on the user intervene. He has to be unaware of all the network problems, in particular in a scenario like the smart home, which aim is to make easier and more relaxing the approach to the everyday life. In AS-Nets such problems have to be tackled in the best possible way, proposing solutions able to allow applications running without dealing with them.

The goal of this thesis is to answer to these problems and to propose an algorithm, that, based on applications needs, facilitates the self-organization of the network and its applications in AS-Nets. Taking the algorithm of Weis, Parzyiegla, Jaeger and Mühl [WPJM06] as starting point, and following all the design phases, its effectiveness and its efficiency have to be validated. Afterwards, it has to be optimized for the following requirements:

Self-Stabilization. Fault-tolerance has to be implemented in the way that the system is able to recover from any possible transient fault. Such property is called self-stabilization and in AS-Nets it represents a crucial goal. In fact, given that the system cannot rely on the user intervene, a fault would heavily affect the system effectiveness.

Heterogeneity. AS-Nets base on the concept of heterogeneity. Actually several types of devices are used, with different capabilities and power constraints. The algorithm has to take care of such peculiarities, in order that such differences do not represent a limitation for the system efficiency. Instead, the algorithm has to get advantages from a differentiated use of resources available in the network.

Devices' Lifetime. In AS-Nets nodes are mostly wireless, then furnished with constrained power. The algorithm has to focus on such point, in the way to minimize the energy consumption in all the aspect of its functioning. From this point depends the effectiveness of the network, and its applicative range as well.

Scalability. As in all the networks scalability is a crucial issue. An algorithm is scalable when, raising the number of nodes, its complexity stays maintainable. In particular in our case complexity is represented by the number of exchanged messages, and by the data stored by nodes. Protocols have to be devised in order to be scalable with respect to them.

1.3 Thesis Outline

Chapter 2 provides a survey of the necessary background to understand the technologies and the protocols treated in this work. First, types of actuator/sensor networks are analyzed, then the discussion passes to analyze possible solutions to build the algorithm. For each technology taken into consideration a short resume of the actual progress report available in literature is discussed.

Chapter 3, based on the considerations made in the previous chapter, proposes an architecture to implement the algorithm. First the choice between the possible solutions is made and argued, then the preferred one is better specified, and the algorithm design is outlined.

Following the main project requirements, *Chapter 4* proposes optimizations of the algorithm identified in the previous chapter. Among the optimization goals, devices' lifetime clashes with stabilization time. In order to find the best trade-off two versions of the algorithm, one optimized for each issue, are deployed. Consequentially, based on the previous considerations a unique algorithm is pointed out.

Chapter 5 describes the implementation phase in order to simulate the algorithm's versions. Onwards first the simulator is chosen, then accordingly the programming language and the program structure. The simulation parameters are identified in order to best study the quality of the proposed optimizations.

Chapter 6 presents the simulations' results. Issues that better characterize the algorithms' behavior are showed and put in relation, in the way to lead the reader to a meaningful and comprehensive analyze of them.

Finally, *Chapter 7* draws the conclusions of this work and provides a survey on the obtained results. Furthermore, points over which future works could focus on are identified, giving an outlook on possible solutions as well.

Chapter 2

Background

In this chapter the technologies used by the algorithm are explained. Beginning with a quick introduction needed to understand their functioning, the state of the art is analyzed as well. First, actuator/sensor networks are presented, their recent growth and future outlooks. In such a network the system has to be structured as a distributed one, and the second section explains why, introducing middlewares as well. In the third section the forms of fault-tolerance are studied, and self-stabilization is defined and explained. A crucial property of the algorithm is the self-organization, and it is argued in the fourth section. Finally, in the fifth section possible solutions to build the communication paradigm are pointed out and analyzed.

2.1 Actuators/Sensors Networks

In recent years research advances in highly integrated and low-power hardware permitted a new category of network to emerge. These are called sensor networks and are composed from an high number of tiny devices (sensors) able to monitor some physical properties of their environment. They are provided with small computation capabilities and with a network interface to perform basic communication. More recently they were provided of a simple wireless interface, creating new applicative scenarios and letting applications to be more *pervasive*. A system is said to be pervasive when it is able to integrate computation into the environment.

In order to equip such a network with the ability not just to detect and sense a property, but also to perform actions based on the sensed inputs *actuators* make their entry in this kind of networks. An actuator is exactly a device that, in response to proper inputs, performs a particular action. Actuator/sensor networks (AS-Nets) extend the sensor networks capabilities, and incorporates in the model design features such as to permit the development of applications able to au-

tonomously perform actions in response to inputs. The applications range results wider meeting automotive and robotics scenarios. Actually, even if actuator/sensor networks are an up-and-coming technology, it is already used in a growing number of contexts.

The devices' power supply is generally provided by batteries which represent a limited energy resource, thus, the optimization of power consumption takes a fundamental importance in application designing. However, in such networks node failures frequently occur. In response to such a topic usually an high number of devices is used. The network should exploit this redundancy to maintain its function, therefore scalability and robustness become crucial issues. Up to now, to deal with limitations of the nodes' capabilities solutions answering to uniform devices have been deployed. In other words the network is usually assumed to be *homogeneous*.

Actually, sensors and actuators are available with different radio coverage, power capacity and processing capabilities, moreover not all devices are mobile. System design has to care about the particular properties of a node, in this way the system can comply with the *heterogeneity* of the network, increasing the performances as well. Especially in AS-Nets the heterogeneity is strictly part of the system. Actuators and sensors are right different device types, furthermore not all the nodes are intended as wireless. Hence some components could be connected to the electric equipment or using a wired communicating interface. It is now plain that an heterogeneous approach to AS-Nets is of crucial importance.

2.1.1 Typical devices

Up to now several device types have been developed. Most used are *motes*, by Intel in collaboration with the University of Berkley [Kli], Shockfish's *TinyNodes* [DFFMM06], Moteiv's *Telos* [PSC05], UCLA's *iBadge* [PLS02] and Scatterweb's *ESB* [SRWV05]. On this field the research advances , and crucial issues on the device are the trade-off between computational power, battery lifetime and dimension. The cost is a decreasing parameter as well, and it is possible to foresee devices with a dimension of some centimeters at a price less then 10$ over the next five years. A very interesting project of the University of Berkley called "Smart Dust" [WLLP01] aims to develop devices of the dimension order of a millimeter over the next ten years. It follows that, by such improvements SA-Nets are going to spread their application area with an impressive rate. Actually devices are commonly provided with a micro controller, usually it is a 16-bit type, with a typical clock of 8 Mhz. As storage memory there are several combinations between Flash memory, common SRAM and EEPROM, providing up to 1024 Kb. The transmission is carried out by a transceiver with common transmission range of about 100 meters in open space. Other interfaces are available, like Bluetooth,serial and

Figure 2.1: An Intel mote

USB. The charging is usually supported by batteries of type AA, AAA or of button type.

2.2 Distributed Systems

The traditional way to structure a network is represented by the centralized architecture. In such a scheme each node of the network is programmed to perform a particular task and communicates with the other ones by message exchanging. The main application program is installed on a prearranged node which, by collecting data from the network, performs the needed actions. In the 1980s distributed systems came up thanks to the greater accessibility to informatics which spread the use of local area networks (LAN) in a very considerable way. A more formalized definition is given by Tanenbaum [TS07]:

> *A distributed system is a collection of independent computers that appear to the users of the system as a single computer.*

By deploying distributed applications, instead of traditional centralized ones, the scalability, the efficiently and the flexibility of systems were significantly increased. As drawback, a considerable time is spent to synchronize and coordinate the different network entities. Therefore, when the distributed approach is not needed the advantage of such a scheme has to be checked. However, for our aims the centralized approach does not fit well, definitively. Actually, in AS-Nets all nodes could be weak, and the need of a master node on which the main program will run represents a critical point for this kind of system. Moreover the need to program

each node in a different way to perform a particular task makes the system difficult to scale. For these reasons the system will be implemented as a distributed system, in which the computation is spread on the whole network.

2.2.1 Middleware

The distributed approach brought also the problem to make different platforms able to communicate among them. Applications must be developed to run without distinction on all the platforms present in the network, leading programmers to a very expensive work of applications adaption. It followed the necessity to build a middleware layer between the operating system and the applicative layer. The middleware layer enables the components to coordinate their activities in such a way that programmers perceive the system as a single, integrated computing facility. Middleware takes a fundamental importance in modern distributed systems,

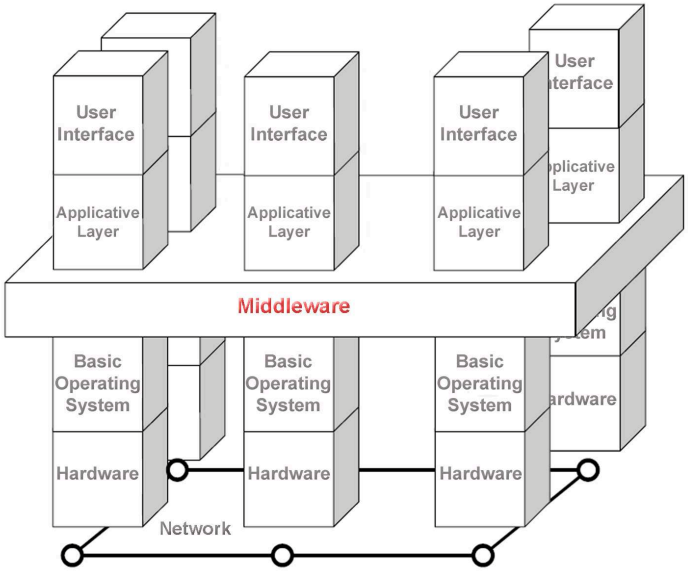

Figure 2.2: Middleware

its main purpose is to manage the communication between components, even if other characteristics like congestion control, concurrency and programming language integration are commonly featured. Some features are designed to improve the efficiency of the system, others to provide an easily usable programming interface. Actually a commercial middleware cannot set aside the usability to assert itself on the market. In WSNs for instance, TinyDB[MHH] is an often used middleware thanks to its tightly coupling with TinyOS[LLWD], the most used operating

system in these kind of networks, but also for the programmer-friendly interface, which permits to investigate easily data disseminated in the network.

In AS-Nets middleware structural characteristics take an importance even bigger. Fault-tolerance becomes a crucial point, as well as power saving and lightweightness. Moreover, for the AS-Nets nature itself, which are designed to mask the network complexity to the final user, the ability to auto-manage changing environments represents a very big improvement. A system able to do this is a system self-organizing. In this work we will concentrate on the middleware just from its structural point of view, which is the one which mostly determines its efficiency. Topics which we address as actually dealing with the middleware's structure are fault-tolerance, self-organization and the communication paradigm.

Fault-Tolerance

Fault-Tolerance is a crucial issue in AS-Nets. A system is fault-tolerant as much as it is able to recover once a failure has occurred. Actually, two approaches to fault-tolerance may be distinguished. The *first* tries to mask faults, in the way that the system can go on working without any service interruption. If this is not enough, and a service interruption happens, the *second* approach tries to recover leading the system to a new legal state. Intervening in two different times, such approaches could be combined in a complementary form, composing a unique two-gears instrument. The goal is to continuously push the system towards a correct state again. In section 2.3 fault-tolerance topic is better argued.

Self-Organization

A growing up topic in middleware deploying is self-organization, which is the ability of the system to organize itself and to adapt its structure to changing environments, all this without user intervention. In particular, self-organizing services perform operations to mask the network complexity to programmers permitting to manage network dynamics. In a system which aim is to keep out the user from the system's working, self-organization is one of the most important challenge for these kind of networks. Self-organization enters in the model design bringing big advantages in AS-Nets, even if, as for self-stabilization, it loads considerably the system. Several services supported by a middleware could be made self-organizing, and in literature some proposals have been already presented. In 2.4 self-organizing services actually dealing with our scenarios are taken into consideration.

Communication Paradigm

Even though, the feature which mostly characterizes a middleware and mostly contributes to its efficiency remains the communication paradigm. The commu-

nication paradigm represents the way by which messages are exchanged in the network. Actually two schemes may be distinguished. In *ID-centric* approach to retrieve data from network a node has to know where such information is located. Afterwards, it can request such data by addressing the interested node. On the contrary in *data-centric* approach, nodes are unaware of data location. Actually messages are routed to right nodes just basing on the characteristics of requested data. In actuator/sensor networks the ID-centric approach is neither feasible neither needed. It is not feasible because of the high dynamism of the network: devices frequently die due to batteries exhaustion and new devices can be deployed as replacement. It is not needed because an application for this kind of network generally is not interested in who actually does the action but just in performing it. For these reasons we speak of data centric systems.

Moreover, in ID-centric communication model, a session for each interacting party has to be maintained. It is clear that in networks composed from thousands of nodes such a scheme doesnt scale at all. Instead, in data-centric networks, the set of interacting nodes is addressed defining the characteristics of requested data. This permits to abstract from single nodes giving a strong advantage when the sizes of the network grow. In section 2.5 the choice of a data-centric communication paradigm is made and properly argued.

2.3 Fault-Tolerance

Solutions to mask faults usually are designed for specific problems, actually, where the system is more exposed to faults instruments of redundancy can be used. Data particularly important could be in part replicated, in the way that if one copy is corrupted, by the other copy it would be possible to obtain the original again. Error correction code (ECC) is an example of such instrument and it is largely used in computer data storage. To check the data coherency proper algorithms are used. Basing on the data transmitted an additional field is filled with a code, in the way that afterwards, by applying the same algorithm on the data, it is possible to compare the result with that field. In such a way it is possible to check the received data effectiveness. For instance, cyclic redundancy checks (CRC) is an instrument of such a type.

Usually such approaches are used to check exchanged messages, and some important stored data. A good point of this approach is that it doesn't need to change the system working, the model design remains untouched and just additional instruments are used. On the other hand a different dedicated solution has to be thought for each possible problem, and anyway it is not ensured that even if an error is detected, it is possible to correct it. Actually, to implement fault-tolerance in a more effective way we need to make the system *self-stabilizing*.

The concept of self-stabilization was introduced by Dijkstra in 1973, he defined a system as self-stabilizing when *"regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps"[Dij74].* In other words by self-stabilization we mean the property of a system to recover from any possible transient fault. A transient fault is an event that may change the global state in a system by corrupting the local state of a node as represented by memory or program counter or by corrupting nodes' connection status or shared memory. Stigmatizing the system as composed of nodes and edges, the union of all the components' state defines the *global state* of the system itself. It is important to underline that the topology of a system is part of its global state, given that it determines the status of the nodes' connections. Moreover, from these considerations it follows that a self-stabilizing system does not need to be initialized. This is a fundamental feature, in particular for ad-hoc networks and dynamic systems.

Self-stabilization incorporates the fault-tolerance topic into the design model, and treats it by a unique vision. This is quite at the opposite with respect to traditional ways, in which every possible fault-cause is treated as stand-alone one by a piecemeal approach. Actually, while they are a tentative to mask errors trying to avoid service interruptions, self-stabilization accepts that an interruption can happen, and guarantees to recover in a bounded time. The two approaches are so complementary, even if only the second one gives the certainty for the system to can go on working. Self-stabilization enters in the model design, addressing new rules in the programming way, and in data and protocols definitions.

2.3.1 Designing for Self-Stabilization

About the design is demonstrated[Sch93] that asymmetry has to be maintained in systems where processes may synchronize among them. This is our case, given that applications could use resources in mutual exclusion. In fact, a node can take simultaneously two roles of two different applications, and an exclusive use of the device's resources have to be guaranteed.

Actually, asymmetry can be maintained in a system by two methods. A system is *asymmetric by state* when all of its component are identical, but they have different initial local states. In this way the system can be realized by identical programs. A system is *asymmetric by identity* when not all of its components are identical. This may be realized by different programs, or by identical programs parameterized by a local id. In general, systems asymmetric only by state cannot be self-stabilizing, in such a case a *central demon* is needed as well[Sch93].

About data and protocol definitions a system can be self-stabilizing only if it is ensured that a memory corruption, a message not delivered, a node failure don't compromise the system working. This could be realized by asynchronous protocols, and by periodical refresh-phases of the system state in each node. In

this way a fault can be discovered, and in case corrected, letting the system to achieve another stable configuration. Evidently cyclic state dependencies have to be avoided during the protocols definition. The program could be stored in a ROM support, in this way it cannot be corrupted.

Furthermore, in the case the wished system is composed by several components, the *fair composition* theory[Dol00] ensures that if such components are self-stabilizing the system that they implement is self-stabilizing too. This point takes a crucial importance in our case, given that the role assignment algorithm we want to build actually bases on other components. Thus, the self-stabilization design of the whole system may be performed just by making self-stabilizing its single elements.

2.4 Self-Organization

In distributed systems the computation organization is a crucial issue and in AS-Nets this point takes an importance even stronger. Usually a configuration phase by the system administrator is needed, which has to set nodes in the proper way, and the network topology as well. Several forms of auto-configurating services coordinated by a proper algorithm are present in literature. In [SAJG00] authors propose a suite of algorithms for self-organize routing and election, other aspects taken into consideration are data aggregation [SK00], coverage [SM01] and lookup services [HMJ05]. Such functionalities do not need of a configuration phase, furthermore they are able to adapt to changing environments.

A different approach, which deals with self-organization by a wider view bases on the concept of *roles*. A role is an application's task possible to carry out by a single node. Applications are analyzed drawing out requisite roles, around which the computation is organized. Once their definition have been done, a proper algorithm is responsible to assign roles to nodes. Römer and Frank [RFMB04] proposed a self-organizing algorithm to assign roles to nodes for a generic task. Such algorithm exploits a rule-based language to define properties a node must have to be assigned for a certain role. In examples provided by authors clustering, coverage and in-network aggregation are topics in which their algorithm find its application. Another project which addresses the same topic is TinyCubus [MLM+05]. The ability of nodes to carry out particular operations is discovered by sending code updates along the network. However, with respect to the previous work any concrete algorithm to self-organize the system has been proposed.

2.5 Communication Paradigm

As argued in section 2.2.1, in AS-Nets the data-centric communication paradigm is the most suitable, definitely. In such a model nodes do not have to know where parts of application are executed, they are actually unaware of their interacting parties. Among the proposed data-centric communication paradigms publish/subscribe stands out thanks to its efficiency for such a network type.

2.5.1 Publish/Subscribe

Publish/Subscribe is an asynchronous data-centric messaging paradigm. The general structure of a PS system is shown in the figure 2.3. Subscribers are nodes interested in some kind of information, they show their intention to receive data issuing a subscription message containing a description of what they are interested in. By issuing an unsubscription message, a subscriber declares he is no more interested in receiving notifications. At the other end of the scheme there are publishers, they produce and diffuse data (events) in form of notification messages. The interaction between publishers and subscribers is implemented by the

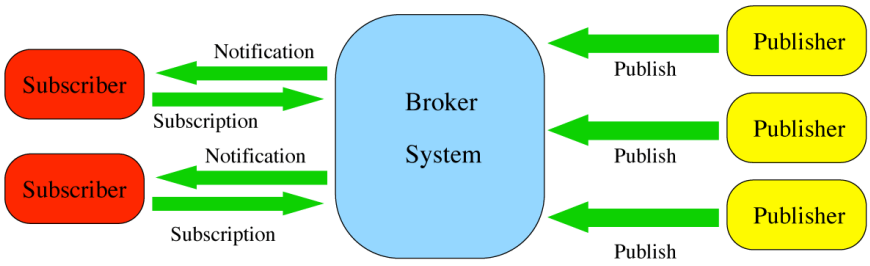

Figure 2.3: Publish/Subscribe System

use of a broker system, which collects and correctly forwards notifications sent by publishers, according to the subscriptions received from subscribers. Actually, publish/subscribe well supports our requirements, in particular by the broker system identities and synchronizations result totally decoupled. And in this way a publisher and a subscriber act knowing anything about each other.

Different Approaches have been proposed. The main difference between them is about the *notification selection*. That is the way by which nodes select the interesting notifications. There are several approaches, and by analyzing them is clear there is a tradeoff between *Expressiveness vs. Scalability*. Actually an higher selectivity of notifications leads to a lower scalability of the system. The most known approaches are:

- *Channel-Based* declare explicitly the notifications' type. This is a simple way, easy to implement and to map on network's address. By associating channels to addresses all things published are delivered without any filtering operations. Of course the expressiveness is considerably limited and producers and consumers are not decoupled. On the other hand the scalability is assured, given that a new notification type can be mapped on one of the pre-existing types, of course decreasing the selectivity.

- *Subject-Based* organize in a hierarchical way the notification types, which are better identified from level to level. For instance, the first level "news" in the second one would be split in "italian news" and "german news", and so on. In this way we gain something on the expressiveness, but producers and consumers are not yet decoupled, and if the tree changes, in order to insert a new subject or something else, the tree representation has to be changed in all the devices.

- *Type-Based* extends the selective potentiality introducing the concept of *event* classes. By that a new data type can be mapped in a pre-existing event class, and so the scalability is preserved. Anyway producers and consumers are still coupled, even if loosely, in fact data structure has to be the same in all the devices. After all this approach is quite the same with respect the previous one.

- *Content-Based* makes the selection by a mathematical expression, composed by *attributes*, *operations*, and *values*. An attribute identifies properties in which a subscriber is interested in, they can be composed by operations which returns a boolean value, and are confronted with fixed values. By forcing to describe desired data with this so fine grain level, it permits to filter out unwanted events. On the other hand such system needs a huge space to store filter rules, thus the scalability is considerably limited.

2.5.2 Broker Systems

As explained in the previous section the broker system characterizes the publish/-subscribe implementation, providing decoupling properties which actually improve the system efficiency. Such system is composed by a set of nodes called *brokers*, connected among them. Each broker is connected to a set of host nodes, from which it collects subscriptions, and to which it forwards published messages if a checking subscription is present. Brokers could be nodes dedicated to carry out such a job, or nodes that besides their tasks provide broker functionalities as well. The ways to implement the broker network, and to forward messages in there, characterize the several proposals available in literature.

Flooding

In the most part of publish/subscribe implementations *flooding* is used. Flooding represents a light-weight solution, given that does not need any communication overhead to keep the system organized. In fact, in flooding-based publish/subscribe, brokers forward received notifications to all other neighbor brokers, and each processed notification is delivered to all local clients with a matching subscription. Hence a coordination between neighboring brokers is not necessary and no additional traffic is generated. Another significance issue is that a subscription becomes active at once as submitted. On the other hand, duplicated messages are sent to the same node, which has to take care to break possible cycles. Furthermore messages are sent to all the brokers, also if some of them have no matching requests. This approach sets aside from energy aware topic as well.

Spanning Tree

The broker network can be implemented in a tree format by using a spanning tree algorithm, which structure the network creating a cycle free connected graph. A broker network structured as a tree gives form to a hierarchical publish/subscribe, as the one proposed by[MJH+05]. The network could be formalized as a graph $G = (N, L)$ where N is the nodes' set, and L is the set of communication links between nodes. A spanning tree $T = (N, L')$ of G is a graph consisting of the same set of nodes N, but only a subset $L' \subseteq L$ of links such that there exists exactly one path between every pair of network nodes. Basically, this means that the graph is connected (there is at least one path between any two nodes). One of the basic theorems of spanning tree states that in a network of n nodes, the tree contains exactly $n - 1$ communication links. The usage of spanning tree to structure the network improves the efficiency of the network protocols, in particular in the case of broadcast messages. In fact, in this case a message is transmitted $n - 1$ times instead of the L times in flooding. Two kinds of spanning trees may

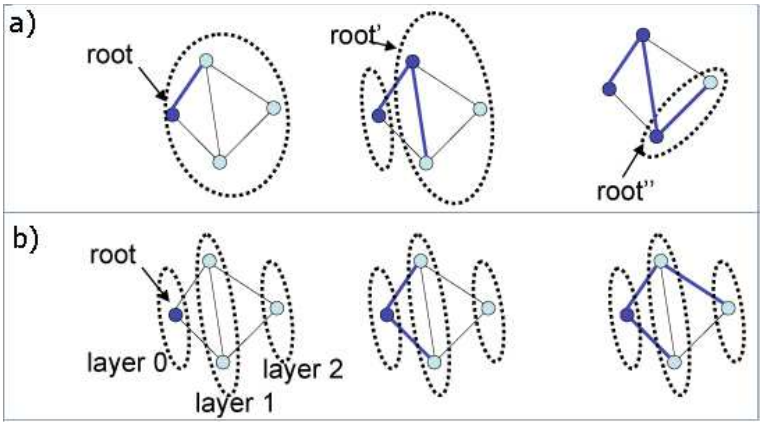

Figure 2.4: Spanning Tree: DFS (a) and BFS (b)

be distinguished, the first is *Breadth-First Search* (BFS), and the tree results from a breadth-first traversal of the underlying network topology. Starting from the root node the tree is built in a parallel way going away from the root (fig. 2.4 b). The second is called *Depht-First Search* (DFS) and the tree is obtained from a depth-first traversal, this means that the algorithm, still starting from the root node, analyzes the set composed by nodes not yet visited, and then step-by-step builds the edge to a new one in a recursive way (fig. 2.4 a).

Rooted spanning tree have a notion of "parent" and they result from the execution of semi-uniform algorithms. An algorithm is called semi-uniform when there is one process which executes a different algorithm. In literature there are several proposals for spanning tree algorithms. Actually they are almost of BFS type, and they difference each other for the way to detect and to break cycles, finally for the way to exchange messages to keep the algorithm operative [Gär03].

Distributed Hash Tables

Distributed Hash Tables(DHT) is a scalable data structure for building large-scale distributed applications. Strongly used in peer-to-peer systems it builds decentralized, self-organizing, fault-tolerant and scalable systems, consisting of symmetric nodes called peers. DHT maps data location to nodes by the use of a particular *key*. Peers have the knowledge of a subset of existing nodes, by such knowledge they are able to recover data from the network, provided that the node depositary is still present. The DHT implementation we take into consideration for this thesis is *Pastry*.

Pastry[RD01] was proposed in 2001, actually several universities participate in such project jointly with Microsoft Research. Pastry assigns to each node a *nodeID* of l bits, that is usually 128-bit long and is interpreted in a 2^b base (usually $b = 4$). For example, with $b = 4$ the identifier 1000111111010 is interpreted as *8FA*. In this way the name space is grouped in three parts and hash keys are equally distributed on it. Each hash key is associated to the node with the closest nodeID value to it, and if two nodes have the same distance the key is replicated in both of them.

To perform routing operations each node stores two data structures: a *routing table* and a *leaf set*. A routing table has $log_{2^b}N$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries in row n refer each to a node whose nodeID matches the present nodeID in the first n digits, but then differs in the $n + 1$th. Moreover, each node stores in its *leaf set* information about nodes with the $l/2$ numerically closest larger nodeIDs, and the $l/2$ nodes with numerically closest smaller nodeIDs.

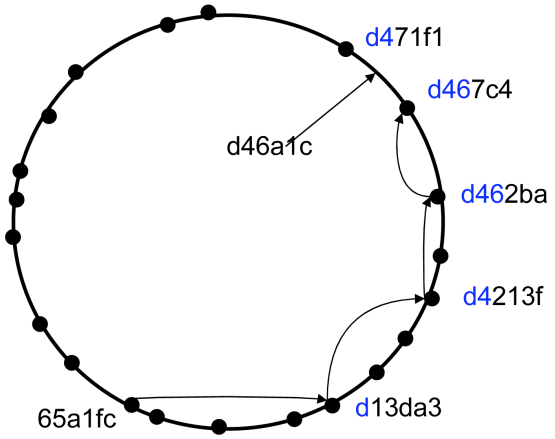

Figure 2.5: Pastry Functioning

Basically to route a message, *node A* checks the presence of the receiver in the leaf set, if it is not there, *node A* compares receiver with its nodeID and with the numerically closer entry in the routing table. If the former is the closest *node A* intends the message for itself, otherwise the latter is chosen as the next hop to deliver the message. In figure 2.5 there is an example on how pastry delivers a message with hash key *d46a1c* starting from the nodeID *65a1fc*.

Messages take $O(log_b N)$ hops on average, where N is the number of nodes in the Pastry network. The factor b is chosen has a tradeoff between the average hops and the routing table complexity. As before said usually $b = 4$. About the fault tolerance the delivery is guaranteed unless $l/2$ or more nodes with adjacent nodeIDs fail simultaneously. In fact, in such a case the leaf set looses its consistency.

Directed Diffusion

Directed diffusion [IGE00] is a particular implementation of publish/subscribe paradigm. Actually, it is not just a broker system, moreover it interpreters in a different way publishers and subscribers as well. Directed diffusion establishes efficient $n-$way communication between one or more sources and sink. As source it is intented every node which produce data, and sinks are nodes interested in. Directed diffusion is characterized by a good level of scalability also in large scenarios. It also well supports fault-tolerance and power-saving features, furthermore it carries out routing, and structures the network as well.

Directed diffusion is performed by the usage of: *interests*, *data messages*, *gradients* and *reinforcements*. By an interest message a node can specifies data in which it is interested in, such node is called a *sink* for the corresponding data type. A data type is expressed by using attribute-value pairs, in this way the expressiveness of the request can be modulated on the node necessity. Interest messages are disseminated through the network setting up gradients between nodes. Specifically, a gradient direction state is created in each node that receives an interest. The gradient direction is set toward the originators of interests along multiple gradient paths and is characterized by a specific interval, which determines the refresh time of the gradient itself.

At the beginning a sink sends an interest message of the appropriate data type with a low refresh interval. Each node that receives an interest message stores belonging informations like sender, interest type and refresh interval in a table, called *interest cache*, and then it forwards the message. Moreover, thanks to the interest cache it is possible to do not create loop in the network. The figure 2.6 shows an example.

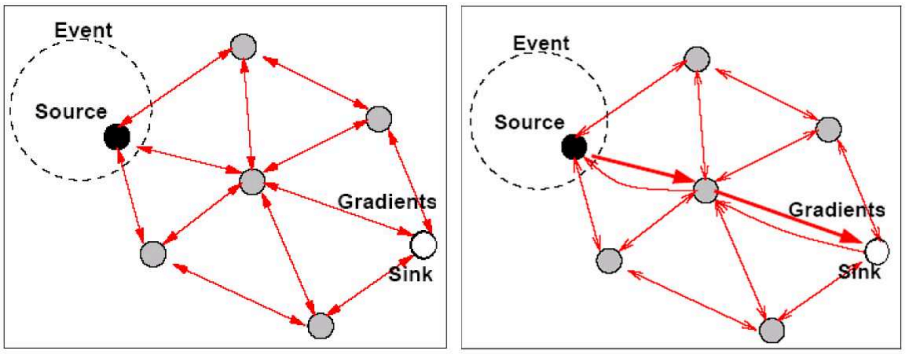

Figure 2.6: Directed Diffusion

When the source node produces a data, it looks in its interest cache, and if a matching entry is present it forwards the message to matching nodes. For

each entry in the interest cache, a data cache is created, in which information about received data messages are stored. When a node receives a data message it analyzes the correspondent data cache, and if that message was already received nothing is done, otherwise the message is forwarded to interested nodes. Each node (source node included) forwards (or sends) data messages following the requested refreshing rate specified in the interest description.

In a bounded time data messages start to reach the sink node by multiple routes. At this point the sink, discovered that a source node for its data really exists in the network, reinforces a particular route (it is sensible to reinforce the first one by which arrived the data message), by sending an interest message with an higher refresh time. Received such message, the selected node performs the same operation, and step by step all the edges of that particular gradient reinforce their refresh times. Now a route with the needed refresh time is created, others $n - 1$ routes are anyway refreshed with a slower rate.

This redundancy is needed to keep effective the network paths, and so to make faster the reaction to topology changes. In fact, when some topology change happens the sink node can reinforce in negative a gradient, selecting another one as the main one. This system seems to be quite expensive, actually each interest has to be refreshed along the network, on the other hand the fault-tolerance is well manageable. Furthermore, given that the network is not structured in a hierarchical way, traffic is distributed with just an inevitable greater stress on the topology center.

Chapter 3

Role Assignment Algorithm

This chapter presents a basic role assignment algorithm, structured as an algorithms stack. The first section shows an overview of the algorithm, pointing out the main features and the quality requirement. Section 3.2 bases on the analyze made in the previous chapter, and accordingly with requirements makes design choices to build the algorithm stack. Then the description goes deeper analyzing the structure of the chosen algorithm stack layer by layer. For each of them the algorithm working is described, with particular focus on the self-stabilizing property.

3.1 Algorithm Overview

This work is part of a project [MOD], which aim is to provide algorithms and tools for self-organizing and self-stabilizing pervasive applications for AS-Nets, especially for a model-driven development. The complete work will be represented by a light-weight middleware able to investigate capabilities needed by applications to run, and accordingly to manage nodes in order to well support distributed computation. Thus, developers will be able to implement distributed applications without having to deal with self-organization and self-stabilization themselves.

The heart of this project is represented by a role assignment algorithm, which, based on the capabilities available in the network, provides to assign roles to nodes. As introduced in section 2.4 a role is an application's task possible to carry out by a single node. Applications are analyzed drawing out requisite roles, around which the computation is therefore organized.

For instance, in the food managing application (see section 1.1) five roles may be distinguished. The fridge inspection, the freezer and larder ones, an internet interface (e.g. the pc), finally a telephonic one (e.g. home phone). Nodes have the knowledge of their capabilities, and they announce them. If in the network enough

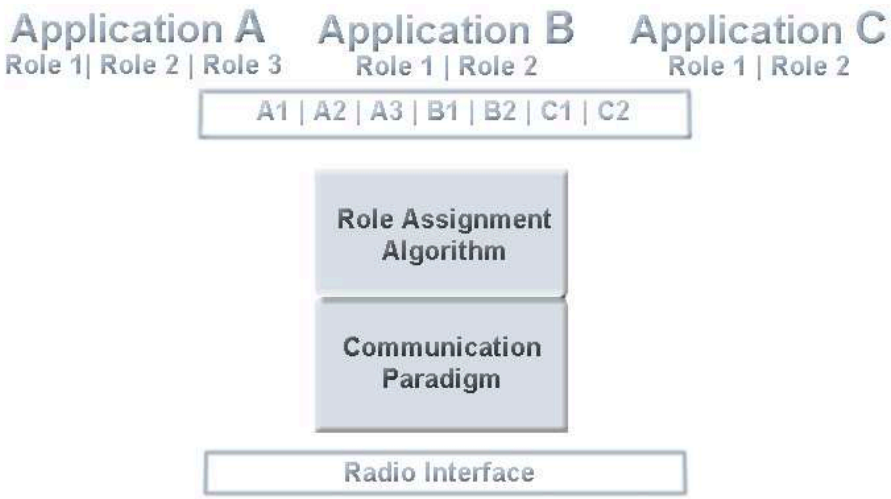

Figure 3.1: Role Assignment Architecture

capabilities are available to run an application the algorithm provides to assign roles. Afterwards, exploiting communication protocols the algorithm features necessary functions to let nodes assigned for a role to interact, thus supporting the application working.

Whether because of redundancy or for multiple possible choices, a role could be carried out by more than one node in the same moment. The algorithm has to choose right one node for each role among the capable ones. In the food managing application the "internet access" role can be assumed by the pc in the study, by the microwave oven and by the smart-phone. Considered the situation in which it has been assigned to the pc, when the user switches it off, the "internet access" role is no more available in the network. At this point the algorithm detects such situation and therefore tries to reassign the role if another capable device is available, for instance the microwave oven. The reassignment is performed without involving the food managing application, which is kept unaware of such reconfigurations, as well as without relying on the user intervene.

The role assignment algorithm is proposed together with communication protocols, forming a **stack of layered algorithms**, based directly on network interfaces (fig. 3.1). This is to ensure that the whole proposed system is optimized for our requirements, which were already introduced in 1.2. Such points represent the criteria by which fitting technologies to build the algorithm can be identified, and they are:

Self-Stabilization Fault-tolerance has to be implemented in the self-stabilizing form. This property represents the first design criteria, and has to be followed during all the implementation phases. This means that the system is able to recover from any possible transient fault. Time spent to lead the system in a new valid state is called *stabilization time*. For the applications efficiency it is important to reduce such time as much as possible.

Heterogeneity Role Assignment algorithm bases on the concept of network heterogeneity. It has to be able to discriminate the different capabilities of nodes in order to assign roles. Furthermore, another aspect of heterogeneity taken into consideration is the different power capacity of nodes. In this way the algorithm can try to load mostly on powerful nodes rather then weaker ones.

Devices Lifetime Devices lifetime is a crucial issue in AS-Nets as already argued in 2.1. Design has to be founded on the devices weakness, therefore protocols and applications have to be planned in order to be power saving as much as possible.

Scalability Issues actually affected by increasing nodes number are devices lifetime and memory consumption. Role Assignment algorithm has to be built in order to minimize their growth with respect to the topology increase.

3.2 Design

To implement role assignment two approaches are feasible: a node can take the role of *leader*, and then it will be responsible to collect capabilities available in the network and to assign roles to capable nodes. Or the assignment can be performed in a distribute way, in which the state of capabilities available in the network and then the assignment are obtained by a close messaging among nodes.

The latter case, even if seems to be more profitable given that the assignment is spread on the whole network, it is very hard to implement. Actually, any self-stabilizing system like that is ever been proposed, and the theory of self-stabilization
citeSchneider:1993:SelfStab explains why: depending on the cases, often that is not feasible.

Hence, role assignment will follow the first approach, and a leader node will be used. Actually some coordinative abilities could be spread over other nodes, however a leader node is needed. Consequentially the flowing assignments scheme could be represented as a first flow in which messages containing capabilities reach the leader node, subsequently a second flow from the leader node sees to assign roles to nodes (fig. 3.2).

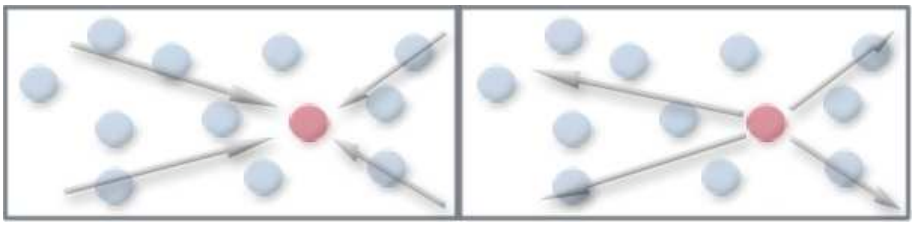

Figure 3.2: Assignments Flowing

The communication paradigm used for this work is *publish/subscribe*. It is data-centric and is particularly suitable for AS-Nets. The approaches proposed in the previous chapter to implement the broker network are: a) flooding, b) distributed hash tables, c) directed diffusion, d) hierarchical pub/sub + spanning tree.

Flooding is too coarse, actually the network is not structured and a message delivery is quite expensive in terms of hops to achieve the receiver. **Distributed hash tables** have the lower hops number to deliver a message, therefore stabilization time would result increased, as well as devices' lifetime. Moreover, being a peer-to-peer algorithm, it loads equally on all the nodes, thus scalability is ensured. However, any self-stabilizing DHT algorithm is present in literature, hence such solution has to be thrown away.

Two publish/subscribe implementations remain among the ones analyzed in the previous chapter, **distributed diffusion**, and the **hierarchical publish/-subscribe** over a spanning tree. The main difference among them is in the way to build the broker network, which in directed diffusion is totally decentralized and builds $n-1$ ways for each interest, reinforcing afterwards the needed ones. While in the hierarchical publish/subscribe the broker network is structured as a tree with $n-1$ links totally, actually with an higher hops number to deliver a message.

In the case of role assignment algorithm the information flowing of the available capabilities in the network has as termination point the leader node, from which another information flow starts again to assign roles. The natural flowing scheme suggests so a **hierarchical approach** to disseminate messages. By this point of view, the latter approach better fits with our aims. In fact, the same routes established to exchange data are exploited to assign roles. Thus, just $n-1$ communications ways are built, therefore routing tables result optimized.

Actually, directed diffusion would build $n-1$ communication ways for each available role only to support the role assignment, plus other $n-1$ for each pair *sink-producer* (see section2.5.2). Therefore, interest cache and data cache would have a higher complexity than hierarchical publish/subscribe's ones. Generally a hierarchical approach is not always the best solution, given that data exchanged

load mostly on nodes as much as closer they are to the top, while in directed diffusion traffic is better distributed. Event though, in this case the better assignments flowing offsets to the asymmetry of load distribution.

However, some interesting points of directed diffusion may be taken into consideration as cue for our publish/subscribe implementation. As in directed diffusion, it is sensible to make the **broker network totally distributed**, spreading such task all over the network. In this way, each node is at the same time host node for its parent, and broker for children.

Actually, we start from the consideration that all the nodes could be weak, and a broker would be provided with higher power capacity due to the higher message exchanged rate. Moreover, the hierarchical publish/subscribe system on which our is based [MJH+05] is self-stabilizing, but not self-organizing. This means that brokers should be initialized in a different way with respect to other nodes. Furthermore, their position should be prearranged in order to form a dedicate broker network. This is totally in contrast with our goals, therefore the dedicated broker network approach is not pursued.

The algorithms stack that comes out follows the one proposed by [WPJM06]. From now onwards, the algorithm described by tizio and caio is taken into consideration as model. In the following sections its functioning is described and explained, while in the next chapter optimizations to it are proposed and analyzed.

3.2.1 Algorithm Stack

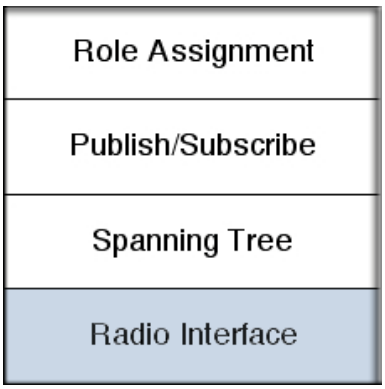

Figure 3.3: Algorithm's Stack

The algorithm stack, which comes out from the previous considerations, is composed as shown in 3.3. The spanning tree structures the network in a hierarchical way, electing the root node as well. Based on this structure, publish/subscribe

implements the routing algorithm disseminating messages following a top-down scheme. Role assignment uses such mechanism to investigate available capabilities in the network and accordingly assigns roles to nodes. In this section layers structure are defined as well as mechanisms to make them self-stabilizing. Interconnections among them are implemented in order to avoid cyclic state dependencies, following the concept of fair composition (see section 2.3.1).

Message structure

The message format is structured in a fixed header plus a variable payload. The first field is *MsgType* and specifies the type of message sent. Two fields contain the information about the *Sender* and the *Receiver*. Finally there is the *Payload*. The MsgType can refer to the spanning tree algorithm (*TreeMsg*) or to publish/-subscribe algorithm (*SubMsg* or *PubMsg*). Role Assignment algorithm exploits publish/subscribe mechanism to communicate. Besides published data, the payload can contain also further information needed by the algorithm protocols.

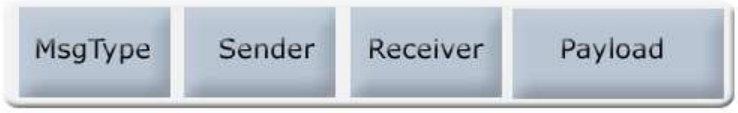

Figure 3.4: Structure of the message

3.3 Self-Stabilizing Spanning Tree

Starting from the bottom the spanning tree algorithm is the first layer of the algorithm stack, it bases directly on the radio interface. In order to allow the use of very simple radio interfaces, the spanning tree requires just elementary radio features, actually is needs only broadcast transmissions, plus a common timer. The preferred spanning tree is of BFS type, in this way the tree is quickly built, and dependencies among nodes are looser with respect to DFS. Following considerations made about self-stabilization in section 2.3.1 the algorithm is implemented as asymmetric by identities, this is realized by the use of the local *ID* of each node. The *root* is elected as the node with the highest ID.

Periodically each node sends a message containing information about the supposed root node, and the *distance* in hops to it. By receiving such messages nodes can check their connection status. Actually, if a node does not receive anymore

messages from the parent, it can suppose to be no more connected to the tree. For this reason messages of such type, which keep network connections alive, are called *heartbeats*. The proposed spanning tree algorithm is an adaptation of the algorithm published by Afek, Kutten, and Yung [AKY91].

Spanning Tree Algorithm

Besides the root information, every node selects a *parent* node as well, that is the node by which it is connected to the root. The parent is chosen as the node with the lowest distance (measured in hops) to the supposed root. At the beginning nodes have no information about each other, so they assume to be the root node and sends an heartbeat. A heartbeat message contains two information in the payload, the supposed root node, and the distance to it. By receiving another heartbeat a node can understand if there is a better root node then itself. If this happens, such node changes its information about the supposed root node, and then chooses as parent the node by which received the message. Afterwards, if an heartbeat with a shorter distance to the root is received, the sender is chosen as new parent node. Accordingly, the distance is updated as well. The heartbeats propagation is performed at once as their reception from the parent. In this way the network is quickly structured, and afterwards updated owing to topology changes. The image 3.5 shows an example.

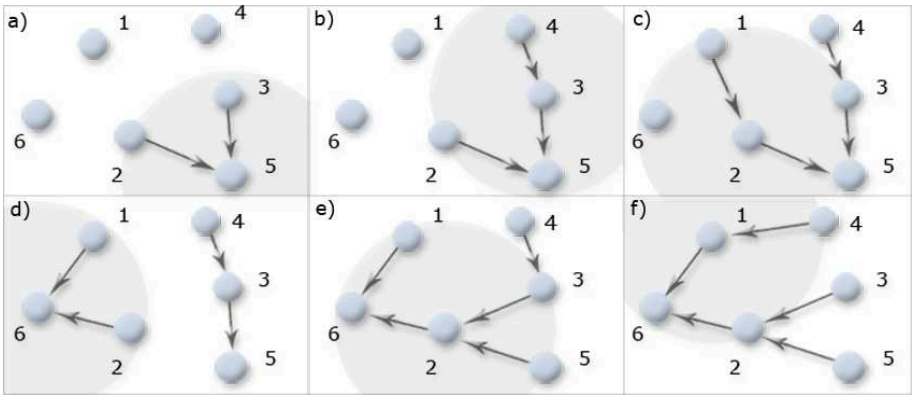

Figure 3.5: Spanning Tree Working

In the first image **(a)** the first node that sends an heartbeat is the node 5. Nodes *2* ($5 > 2$) and *3* ($5 > 3$) lying in its transmission range, chooses node *5* as root and as parent as well. Then the node *3* forwards the heartbeat **(b)** and the node *4* takes *5* as root too ($5 > 4$), assuming node *3* as parent. In **(c)** node *2* forwards the heartbeat causing node *1* to choose node *5* as root ($5 > 1$) and node

2 as parent. Node *6* goes on thinking itself as root ($5 < 6$). In **(d)** node *6* sends an heartbeat taking at once node *1* and *2* as children. Nodes *3* and *5* assumes *2* as parent and *6* as root when the heartbeat by the node *2* comes **(e)**. Finally, **(f)** node *4* consequentially to the *1*'s heartbeat receiving takes it as parent and *6* as root. In fact node *1* is a parent with a shorter distance to the root (1 hop) with respect to the node *3* (2 hops).

Self-Stabilization

To support self-stabilization nodes' state has to be continuously refreshed, this is achieved by the periodically sending of heartbeats. In particular, heartbeats are issued by the root node, and subsequently forwarded from parent to children.

By heartbeats reception nodes can continuously check the tree coherency, and so be sure to be still connected to the network. When a node does not receive the heartbeat anymore it resets itself, assuming itself as root node. In particular the period following which the root node issues an heartbeat transmission is determined by a timer running out. Supposed that nodes set their timeout at δ, root node needs a shorter timeout, in the way to send the heartbeat before the children' timer running out.

Furthermore, in order to improve the system fault-tolerance it is sensible to allow a certain flexibility margin with respect to possible missed deliveries. Thus, the root node will set its timeout to $\delta/3$, in this way a normal node would receive two heartbeats in a timeout period. Finally, if the root node fails, after δ seconds since last heartbeat reception, nodes' timer run out. Hence, they reset themselves and issue a heartbeat. In such a way the algorithm rebuild the tree and the node with the highest ID, among the nodes still working, is elected to be the new root node.

3.4 Hierarchical Publish/Subscribe

The publish/subscribe algorithm is the second layer of the stack. Using the edges of the tree built by the spanning tree algorithm it exports a communication mechanism in the form of a hierarchical publish/subscribe system.

Publish/Subscribe works on the same hierarchical structure of the spanning tree , and so the root node becomes the center of the messages' dissemination as well. The publish/subscribe type chosen is between the subject-based model and the type-based one. This choice comes out as trade-off between scalability and expressiveness, as pointed out in [MJH+05], moreover it is motivated by the data type used by the role assignment algorithm. Tasks and correlations among them are in fact already structured based on the concept of role. It follows that the

great expressiveness of the content-based model is unnecessary and would bring unjustifiable scalability constraints. In literature proposed works to implement a self-stabilizing hierarchical publish/subscribe system are present, and the publish/subscribe adopted in this work bases on [MJH+05].

Publish/Subscribe Algorithm

Nodes subscribe for the message types in which they are interested in through sending a **SubMsg** to the parent. The parent at his turn subscribes to his parent with its own subscriptions combined with the subscriptions of its children. By the subscriptions combination, each type of them is sent just once behalf of the sender, thus scalability is pursued. Subscriptions bubble up along the edges of the tree until they reach the root node, and on their way nodes store arrived subscriptions in a routing table. Thus, nodes know their children's subscriptions, and the root node has a complete view of the subscription's state. In 3.6 nodes *4*, *3* and *5* send

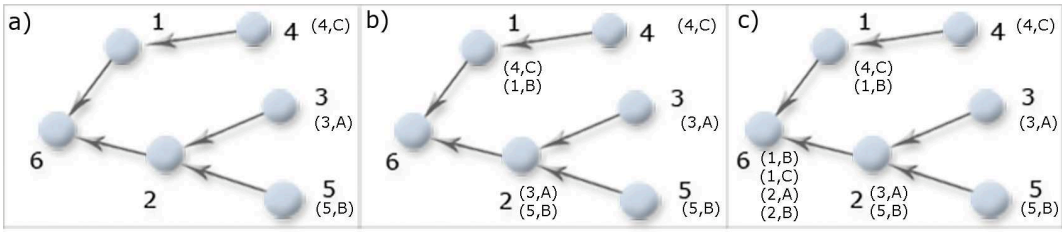

Figure 3.6: Publish/Subscribe

their subscription to parent **(a)**, which combine them with their own subscriptions, sending the result to root node**(b)**. Finally, root node has the complete view of the subscriptions in the network **c**.

The publish/subscribe exports three functions:

- *Subscribe(PSType)*
- *Publish(PSType,message)*
- *PublishSingle(PSType,message)*

The first function subscribes the node for message type in which it is interested. The second one is used to publish a message of a certain type, that will be received by all the nodes subscribed in that particular PSType. Published message bubbles up till the root node, which routes it towards children with a matching entry in the routing table. The last function publishes a message of a certain PSType too, but the receiver will be only one, and is chosen by using an hash function among the children with a matching entry in the routing table.

Self-Stabilization

To support self-stabilization the approach consists in making routing entries just *leased*. This means that after a leasing period π, if an entry in the routing table is not renewd it is erased. To support such a mechanism a *second-chance* algorithm is used. Therefore, in addition to the *ID* of the Subscriber and to the *PSMsgType* for which it made the request, another field it is used to check the entry's age. Such

ID	PSMsgType	TTL
3	B	1
45	G	2
23	S	1

field is called *TimeToLive* (TTL), and each time that a subscription is inserted or refreshed the corresponding TTL field is set to *1*. After a time of π, all the TTL fields are checked, if someone is already set to *0* the corresponding entry is erased, otherwise it is set to *0*. Thus, to unsubscribe from a certain message type, nodes can just stop to renew its subscription. In this way not only publish/subscribe system is able to recover from internal faults but also from certain external faults. For example, if a client crashes, its subscriptions are automatically removed after their leases have expired.

About the leasing period π, it is sensible to do not count it as an absolute time, but in relation with received heartbeats. Hence, TTL fields are checked every heartbeat reception, and they address how many heartbeats the subscription can stay alive without being renewed. This is to set aside non-critic problems of the layers below, therefore a single one missed heartbeat or transmission delays do not affect publish/subscribe.

3.5 Self-Stabilizing Role Assignment

The role assignment algorithm is at the top of the stack, and bases on the concept of roles. A role is an application task able to be carried out by a single node. Applications are split into several collaborating roles; accordingly nodes capabilities are analyzed in order to identify roles which a node is able to carry out. Using the functions exported by publish/subscribe nodes announce their capabilities, and based on them role assignment assigns roles to nodes. Afterwards it continuously checks their activation, and in case a node assigned for a role faults the algorithm detects it and provides to reassign that role.

As argued in section 3.2, a node is chosen as leader, such node has the task to collect nodes capabilities and then to assign roles. Considering the hierarchical

structure built by spanning tree and on which the publish/subscribe works, it is sensible to choose the root node as leader for our algorithm.

Assuming $r_{x1}, r_{x2}, r_{x3}, ...$ as the roles set R_x for the application X, and Z the number of applications, the roles' set **R** of all the applications is:

$$R = \{R_A \cup R_B \cup R_C \cup ... \cup R_Z\} = \{r_{a1}, r_{a2}, ..., r_{b1}, r_{b2}, ..., r_{c1}, ..., r_{z1}, ..., r_{zm}\} \tag{3.1}$$

Roles are mapped into PSTypes as a combination of a specific role and a **flag** that specifies a particular meaning for the algorithm. The flag can be of two values: *possible* or *active*. Hence each role refers to two publish/subscribe message types and so: $PSType \in \{r_{zm}|flag\}$. Assuming as S_y the roles that the node y can take, the whole set of available roles in the network **D**:

$$D = \{S_1 \cup S_2 \cup S_3 \cup ... \cup S_N\} \subseteq R \tag{3.2}$$

where N is the number of nodes in the network. As P_y we also refer to the set of roles the node y took. In the case all the roles are correctly assigned it will result:

$$R = \{P_1 \cup P_2 \cup P_3 \cup ... \cup P_N\} \tag{3.3}$$

Role Assignment Algorithm

The role assignment algorithm consists of six phases. It can find itself in a different phase of the first five for each application running on the system. Actually, an application might have all its roles successfully running, while another not. Only when the algorithm reaches the fifth phase for all the applications it passes to the sixth.

In phase 1, devices analyze their own S_y set of roles able to perform, and through publish/subscribe they send a SubMsg subscribing for message types corresponding to such roles with the flag set to *possible*. In this way roles availabilities bubble up from child to parent, and each node subscribe for their own availabilities plus the ones received from children (S_{RCV} set). Thus, node y sends a SubMsg subscribing for:

$$PSTypes = \bigcup_{r_{xm} \in (S_y \cup S_{RCV})} (r_{xm}|possible) \tag{3.4}$$

Following the publish/subscribe mechanism subscriptions bubble up to the root node, which has the complete view of subscriptions in the network, and then the knowledge of the available roles set D (fig. 3.7 **a**).

In phase 2, the root node compares the D set with the R_z sets of roles into which each application is split. If a $R_x \subseteq D$ this means that there are available nodes for all roles of application x. Therefore, the algorithm passes to phase 3, otherwise it stops without performing any other operation.

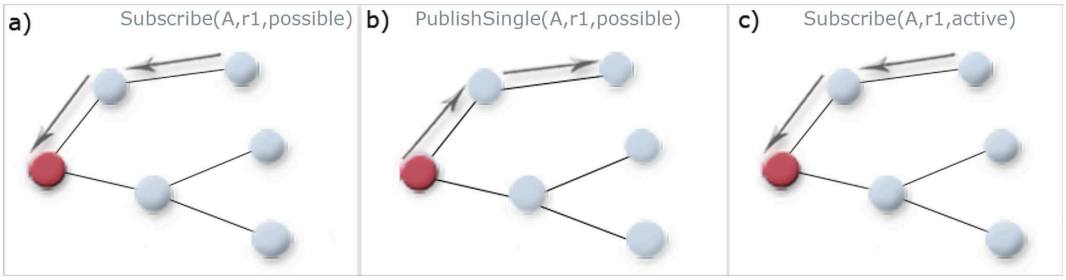

Figure 3.7: Role Assignment

In phase 3, the root node assigns roles $\forall x \in Z | R_x \subseteq D$. In other words it assigns roles for each application which roles are available in the network. It is done by sending a publish-single message exactly to one node for each role. The PSType is $(r_{xn}|possible)$ (fig. 3.7 **b**). In the payload we have to put another information about the meaning of such publish-single message, and so there is a field *operation* set to *activate*.

In phase 4, nodes chosen for a role receive a publish-single message, and if the *operation* field in the payload is set to *activate* they understand to have been assigned for such roles. Thus, they reply subscribing for the message type $(r_{xn}|active)$ (fig. 3.7 **c**). From now on, nodes will subscribe for such message types:

$$PSTypes = \bigcup_{r_{xm} \in (S_y \cup S_{RCV})} (r_{xm}|possible) + \bigcup_{r_{xm} \in (P_y \cup P_{RCV})} (r_{xm}|active) \qquad (3.5)$$

where P_{RCV} is the set containing the children's subscriptions for activated roles.

In phase 5, applications, which roles have been successfully assigned, are correctly running and to send a message it is enough to publish a message of the type $(r_{xn}|active)$. When all the applications find them selves in such a phase the role assignment algorithm passes to phase 6.

Phase 6 is the active status of the algorithm. In the case that a role would result missing, the algorithm returns to the second phase. Moreover, it might happen that two activations for the same role are detected. In such a case the node that has realized it sends a message to children telling to remove such activations and the algorithm passes to phase 2.

Self-Stabilization

To make the role assignment algorithm self-stabilizing is used the same approach of publish/subscribe. Therefore, the information stored about the available roles along the network have to be continuously refreshed. Actually, being roles mapped into publish/subscribe message types, such mechanism is automatically performed. Periodically nodes subscribe for message types corresponding to their availabilities, and to the roles for which they are assigned. In this way a failure of a node assigned for a certain role can be detected in a short time span, and accordingly the root can reassign such role. If a node able to carry out a role is not available in the network the algorithm lays in the phase 5, in which some applications are running, and some others not correclty. This is anyway a form of fault-tolerance: if a role is missing the whole system is not blocked.

Chapter 4

Optimizations

The algorithm shown in the previous chapter answers to our requirements. However, section 4.1 shows how it is not optimized for them. Among such requirements two result in contrast: devices' lifetime and stabilization time. The algorithm optimization is pursued by taking one of the previous criteria at a time, and for each of them an extremely tuned algorithm version is deployed. The trade-off between the two versions is afterwards made jointly with the optimizations dealing with the other requirements.

In section 4.2 mechanisms to reduce the energy consumption are pointed out. Therefore, the algorithm is shown in pseudo-code and commented. On the contrary, section 4.3 analyzes and implements ways to speed up the stabilization time, showing the differences with the previous version's code as well. Finally, section 4.4 proposes a unique algorithm that, merging the devices' lifetime and the stabilization time versions, is also optimized for another important requirement: the network heterogeneity.

4.1 Optimizations Overview

The algorithm described in the previous chapter follows the one proposed by [WPJM06], and well fits with the requirements pointed out in1.2. However, in order to find a fine-tuned trade-off between energy consumption and stabilization time, a deeper analyze is suitable. In the algorithm presented in the previous section several points are object of trade-off.

Looking at the algorithm from the publish/subscribe point of view, to improve the stabilization time faster methods to build routing tables would be needed. Furthermore, the role assignment algorithm would exploit mechanisms of fast events notifications, in the way to make the algorithm in condition of stabilize in a shorter time span. However, such improvements would use a bigger rate of

exchanged messages, leading to an higher energy consumption. During the algorithm working stabilization phase is the most expensive in term of messages exchanged. Actually, a more regular temporal traffic shape would help the algorithm to minimize the number of collisions, saving on useless configuration phases and on energy consumption as well. Even though, to realize it rules to limit the messages transmission should be adopted, lowering the stabilization time.

In order to find the best solution for our role assignment algorithm, allowing it to better reply to the AS-Nets challenge, optimizations are performed by a single point of view at a time. This means that two version will be implemented, one focused on energy consumption, the second one on the stabilization time.

4.2 Energy Consumption

The issue which mostly affects energy consumption is the messages exchange (both transmitted and received). Thus, to increment the devices' lifetime this point has to be reduced as much as possible.

4.2.1 Message Piggybacking

Figure 4.1: Devices' Lifetime Message Format

A solution could be to **piggyback** information of different protocols in the same transmitted message. In particular, a point in which it is possible to use such operation is in the periodically messages exchange to refresh data structures. Actually, nodes receive and forward heartbeats, afterwards they send a SubMsg to their parent to refresh subscriptions. SubMsgs and heartbeats can be englobed in one, piggybacking the heartbeat on the SubMsg. To make this the message format has to be changed. Information about the supposed root node and the distance to it has to be present in every message. The new format is in the figure 4.1.

4.2.2 Heartbeats Decoupling

Heartbeats are usually transmitted at once they are received, this means that nodes geographical close forward the heartbeat in the same time span. Hence, the collisions probability in such time span is considerably higher. In our case, in which the communication is not reliable, a collision is very expensive in terms of system efficiency. Actually, two sequential collided heartbeats bring underlying nodes to thing broken the link to the parent.

A possible approach to reduce the collisions probability is to **decouple** the heartbeat forward from its reception. This means that it is not forwarded at once it is received, but after a time independent from all the other nodes. This can be done by using a timer initialized randomly. By reducing the collisions number, useless configuration phases are avoided as well. This point assumes a significant importance, given that during a configuration phase nodes send more messages, therefore using more energy.

4.2.3 Count to Infinity Problem

Expedients introduced in this section actually improve the devices lifetime, but introduce some problems as well. In figure 4.2 the count to infinity problem is shown. Numbers next to each node represent the supposed root node and the distance to it. Moreover, each node has got a timer set with the same time span but in different moments. In (**a**) the node *5* faults, and after a certain time span

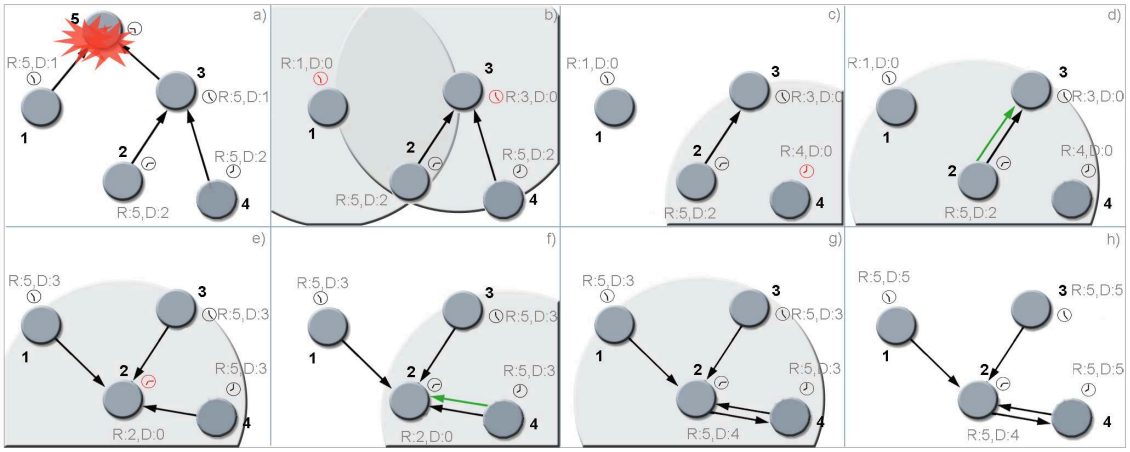

Figure 4.2: Count to Infinity Problem

node *3*'s timer runs out, in particular such time corresponds to $(\delta + delay_{3-5})$ after the last heartbeat reception, where δ is the timeout period (see sec. 3.3), and

$delay_{3-5}$ is the difference between the *3*'s timer and the parent's one. Also the *1*'s timer runs out and so both nodes reset themselves (**b**). In (**c**) also the node *4* reset itself, so it is passed a time corresponding to ($\delta + delay_{3-5} + delay_{4-3}$). But node *2*, which $delay_{2-3}$ is longer with respect to $delay_{4-3}$, is still connected to node *3*, thinking still alive the root node *5*. At (**d**) just before its timer running out, node *2* sends a message.

Given that all the messages are now used to build the tree, other nodes, by receiving a message containing root node *5*, assume node *2* as parent (**e**). In the picture (**f**) node *4* sends a message and node *2*, by receiving a message containing as root node *5*, takes node *4* as parent (**g**). At this point a chain it is formed, with all the nodes that are thinking as root node a faulted node. Furthermore, nodes *1,3* and *4*, increment the supposed distance to the root (**h**), and node *2* does the same given that its parent is node *4*. Thus, the chain bring nodes to increment the distance value, message after message, up to count to infinity.

To solve this problem, spanning tree algorithm has to be changed in order to do not permit such an occurrence. In particular the only solution is that nodes which timer runs out, before to choose a new parent with a greater or equal distance to the previous root node, have to wait for a certain time span. During such interval, timers of nodes in the same transmission range, and with the same or inferior distance to the previous root node, will run out definitely. Such interval has to be set to two timeout periods. Actually the first is needed to let timers run out in normal environments, the second consider the case in which the node lost three regular heartbeats before the parent's reset.

4.2.4 The Code

The starting-point of the algorithm is represented by the **Recv()** function (list. 4.1). Such function is called to manage a message reception. As first the function *Count_to_Infinity_Problem* sees to manage the homonymous effect described in the previous subsection. If such function returns a positive value the computation goes on, otherwise the message is discarded (cf. line 2).

Going on the message is analyzed by the spanning tree, which compares the root and the distance field of the message with the node's ones. If a better root (cf. line 3) or a parent with a shorter distance to the root (cf. line 10) it is detected, node's fields *root_*, *parent_* and *distance_* are updated. The timer is reset and an heartbeat is sent as well. Otherwise, if the sender is the node's parent (cf. line 16), the flag *recv_* is set, and the distance is updated as well. *Recv_* is read by the *Time_out()* to know if an heartbeat was received. By renewing the *distance_* field such information is kept effective in the case the path to the root changes.

```
1  void Recv(message mes){
       if( Count_to_Infinity_Problem(mes.sender, mes.distance)){
3          if(mes.root > root_){
               root_←mes.root;
5              parent_←mes.sender;
               distance_←mes.distance;
7              Timer(timeout);
               Send(TreeMsg);
9          }
           else if(mes.root=root_ && mes.distance < distance_){
11             parent_←mes.sender;
               distance_←mes.distance;
13             Timer(timeout);
               Send(TreeMsg);
15         }
           else if(mes.root=root_ && parent_=mes.sender){
17             recv_←0;
               distance_←mes.distance;
19         }
           if(mes.root=root_ && (mes.receiver=here_ ||
21             (mes.receiver = CHILDREN && mes.sender=parent_) ) ){
               OnProcessMessage(mes);
23         }
       }
25 }
```

Listing 4.1: Receive

Finally, if the message is intended for the node itself the spanning tree algorithm passes the message to the publish/subscribe, by calling the *OnProcessMessage* function (cf. line 22). This happens if the receiver field contains exactly the node's address, or a special address, which generically indicates the children, and the sender is the node's parent.

The **OnProcessMessage()** function (list.4.2) is responsible to process received messages. A message can be a *SubMsg*, or a *PubMsg*. In the first case (cf. lines 2-12) it contains child's subscriptions, and they have to be inserted in the routing table, or simply updated if already present. In the routing table the update is made by setting the TTL value of the corresponding entry. If the *SubMsg* has the flag set to *ReplyingSingle* it is a subscription telling that a child is accepting a role. Thus, it has to be forwarded at once without waiting for the timer running out, like usually is done for normal subscriptions. If the message

```
26 void OnProcessMessage(message mes){
     if(mes.type="SubMsg"){
28     if((mes.type,mes.sender)∈(S_here ⋃ S_RCV)){
         RefreshSubscription(mes.type,mes.sender);
30     }
       else{
32       InsertNewSubscription(mes.type,mes.sender);
         if(DecodeFlag(mes.flag)="ReplyingSingle")
34         Send(SubMsg,mes.flag);
       }
36   }
     else if(mes.type="PubMsg"){
38     if(DecodeFlag(mes.flag)="DataMsg"){
         if(mes.type ∈ P_here)
40         OnConsumeMsg(message mes);
         else if(mes.type ∈ P_RCV)
42         Send(PubMsg,message mes);
       }
44     else if(DecodeFlag(mes.flag)="SingleMsg"){
         if(mes.type ∈ S_here){
46         OnActivateRole(mes.type);
           Send(PubMsg,EncodeType(mes.type,"Active"),
48                                   EncodeFlag("ReplyingSingle"));
         }
50       else if(mes.type ∈ S_RCV)
           Send(PubMsg,mes.type,mes.flag);
52     }
       else if(DecodeFlag(mes.flag)="Remove"){
54       if(mes.type ∈ P_here || mes.type ∈ P_RCV){
           Remove_Roles_Activation(mes.type);
56         if(mes.type ∈ P_RCV)
             Send(PubMsg,mes.type,mes.flag);
58       }
       }
60   }
   }
```

Listing 4.2: On Process Message

is a *PubMsg* it can contain data or transporting particular meanings of the role assignment algorithm. The algorithm understands the difference thanks to a flag (*mes.flag*) inserted in the payload. In the first case, if the node is subscribed for that message type it is analyzed by the *OnConsumeMsg()* function(cf. line 40),

otherwise, if a child subscribed for it exists, it is forwarded to children. Data message are sent only to nodes featuring a role, and so the *mes.type* is searched just in P sets. Furthermore, given that a role is featured in the same instant only by a node, the message is forwarded only if the node itself is not subscribed for it. A published message is also used to assign roles to nodes (*Publish Single* ones), or to say to nodes to remove the activation for a particular role. In the former case(cf. lines 44-52), if the message is intended for the node itself the role is activated and a *SubMsg* with a particular flag is instantaneously sent towards the root node. The particular flag is important in order to say to parents to forward it at once. Otherwise, if a child is subscribed for it, the message is forwarded down exactly to it. Finally, if the *mes.flag* is *Remove* (cf. lines 53-57), nodes have to remove the activation for the type contained in *mes.type*, if some subscription it is present in the routing table for that message type, the routing entry is erased, and the message is forwarded down.

The function **Time_out()** is called by the timer, when it runs out. At the beginning this function checks if in the last two timeout periods (2*δ)the node's changed root node. If yes it decrement a counter, in the way to let the node free

```
62 void Time_out(){
     Timer(timeout);
64   Count_To_Infinity_Problem(set);
     if(root_≠ here_ && recv_< limit_){
66     recv_++;
       OnHeartbeat();
68     Send(SubMsg);
     }
70   else if(recv_=limit_ && here_≠ root_){
       Count_To_Infinity_Problem(reset);
72     Remove_Roles_Activations();
       recv_←0;
74     root_←here_;
       parent_←here;
76     distance_←0;
       Send(TreeMsg);
78   }
     else if (here_=root){
80     Prepare(TreeMsg);
       OnHeartbeat();
82   }
   }
```

Listing 4.3: Time Out

when such a counter runs out. Such operations are performed by the function *Count_To_Infinity_Problem()*, with the parameter *set*. Such a call is different with respect to the one in the *Recv()* function, where the parameters are two. For all the nodes which are not the root node, such a function has to check if an heartbeat was received during last *limit_* times it ran out. If this happened, the *OnHeartbeat()* function is called, which updates the routing table, then the heartbeat and subscriptions are forwarded in a *SubMsg*. If the node has any subscriptions, and the routing table is empty, just a TreeMsg is sent. If during the last *limit_* times the timer ran out any heartbeat was received, the node reset itself. Hence, the recv_ counter is reset, the root node is assumed as the node itself, and the parent node as well. The root node just updates the routing table and sends an heartbeat.

The **OnHeartbeat()** function performs the update of the routing table. This means that the TTL field of every entry is decremented. If such a value is zero, the corresponding entry is erased from the routing table. Finally, once the routing table is rebuilt, the *RoleAssignment()* function is called.

The **RoleAssignment()** function has two different behaviors, depending if the

```
84 void OnHeartbeat(){
     Age_Routing_Table();
86   Remove_Old_Entries();
     RoleAssignment();
88 }
   void RoleAssignment(){
91   for(∀(x,m) | r_xm ∈ R){
       if(∃ y,z ∈ P_RCV | y ≡ r_xm & z ≡ r_xm){
93       Remove_Roles_Activation(r_xm|active);
         Send(PubMsg, EncodeType(r_xm, "Active"),
95                              EncodeFlag("Remove"));
       }
97   }
     if(here_=root_){
99     for(∀x | R_x ∈ R)
         if((S_RCV ⊇ R_x)
101        for(∀m | r_xm ∈ R_x)
             if((r_xm ∈ P_RCV))
103            Send(PubMsg, EncodeType(r_xm, "Possible"),
                                    EncodeFlag("SingleMsg"));
105  }
   }
```

Listing 4.4: OnHeartbeat and Role Assignment

node is the root one or not. If the node is not the root, just the first part of the function is executed. In particular it checks the routing table, if more subscriptions for the same role with the flag *active* are present, it sends a message down to say to remove the activation of that particular node. If the node is the root node, it has to assign roles too. If subscriptions for all the roles of an application are present, for all of them a single message is sent to one of the child who subscribed for that. The choice of a particular node is performed randomly.

4.3 Stabilization Time

Assuming the system in every possible status, the *stabilization time* is the time the algorithm spends to bring the system in a status in which every role is correctly assigned. The longest time to stabilize takes place during the first configuration phase, in which none role is yet assigned. Stabilization time is time lost for the system, and it is important to make it as shorter as possible.

By such a point of view all the considerations made in the previous section change definitely. Actually, it is very expensive to support heartbeats decoupling. In fact, a node can be forced to wait two heartbeat periods before to stabilize. In this section those solutions are put aside, and as starting point the original algorithm in taken into consideration.

A point on which we want to intervene is the time spent to keep effective the routing table. In particular, the modifications have to aim to make the role assignment algorithm aware of the real subscription status as faster as possible.

4.3.1 Fast Notifications

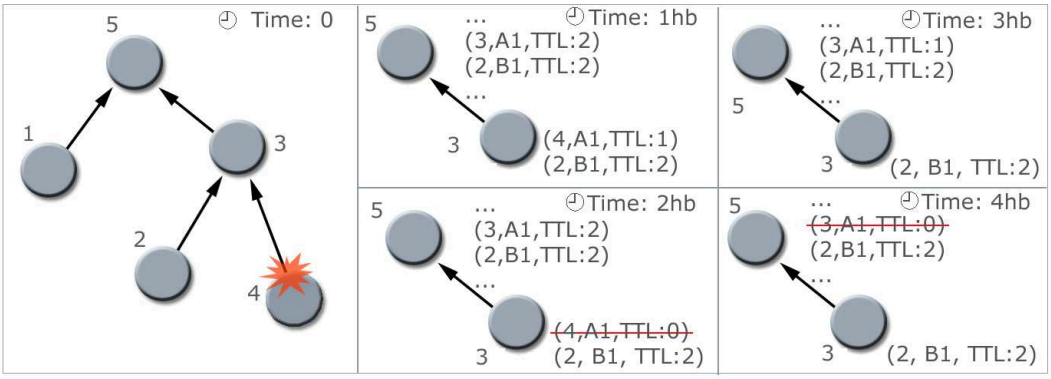

Figure 4.3: Notifications Delay

The time the algorithm spends to stabilize is affected from the time a new information bubble up towards the root node, definitely. Information concerning with the stabilization time are: **a)** New availabilities found, **b)** Faults detection. The first type is typical of topology changes and it describes the time that a node's subscription spends to bubble up towards the root node. In particular, when the algorithm is not yet in the active phase, maybe because an availability misses, stabilization time can depends in a strong way from this parameter.

Assuming that any heartbeat is missed, and ignoring the transmission delays, a change in a node's routing table is learned by parents only after a time of two heartbeat periods ($\frac{2}{3} * \delta$), for level. The **problem** is that if a role is no more active, for instance because the relative node at the fifth level turned off, the root node needs $\frac{10}{3} * \delta$ seconds before to be aware of it. In figure 4.3 the fault notification delay is better pointed out. Node *4*, which is assigned for role *A1*, faults and the root registers it only during the following fourth heartbeat. Such delay definitely affects unjustifiably the stabilization time, therefore has to be strongly reduced.

A **solution** to speed up such operations is that a node which removes a routing entry or inserts a new one, sends a particular message to communicate it to the parent. It has to be done just if any other entry of the same type is present among its and children's subscriptions. Other nodes along the route can update their routing table as well. In this way a new availability can be notified in just one heartbeat period. To make the root aware of a role missing, just $\frac{2}{3} * \delta$ seconds plus the delay to deliver the message are needed. This means that in a time less than δ the root node can provide to reassign a missed role.

4.3.2 Double Activation Problem

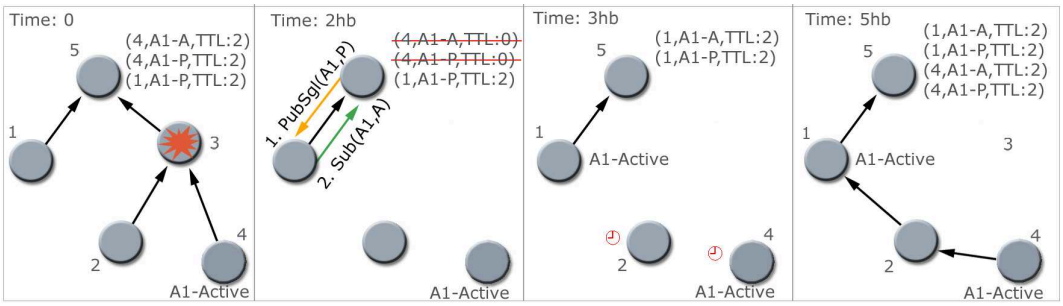

Figure 4.4: Double Activation Problem

When two (or more) activations of the same role are detected, the algorithm intervenes telling to the interested nodes to deactivate that particular role. Such

operation is needed to ensure a correct algorithm working, but in terms of time spent is considerably expensive. Hence, when a situation that can bring to such a situation happens, a mechanism to avoid a double activation should be adopted. A particular scenario in which a double activation happens definitely is the one in figure 4.4. Node *4* is assigned for role *A1*, and node *3* faults. Assuming that any heartbeat is missed, and ignoring the transmission delays, the faulted node's parent (in this case the root) is aware of the fault after $\frac{2}{3} * \delta$ seconds. Consequentially it opens a new assignment for the missed *A1* role. After δ seconds since the failure, the faulted node's children reset themselves, choosing another path towards the root node.

The **problem** is that When the *4*'s subscriptions bubbles up to the root a double activation is detected. On the whole $2 * \delta$ seconds are spent and besides the first $\frac{2}{3} * \delta$ ones, necessary for the root to detect the fault, others are actually wasted.

The **solution** consists in removing the roles activation interested by the fault. When a node reset itself, it has to remove its activations, moreover, it has to send a message to children telling to make the same. In this way after δ seconds from the node's fault, the role is correctly assigned again.

4.3.3 Role Reassignment

In order to further speed up the stabilization time, every node should be able to assign again a missed role. Actually, a node along the path between the root node and the faulted node, which has the capability to take that role or it has a matching entry in the routing table, has all the information needed to perform such operation. In this way if ξ is the delay to transmit a message and A is the node which could reassign the missed role, a time $2 * (\xi * hops_{A->root})$ is saved. The first $\xi * hops_{A->root}$ term refers to the fault notification, while the second one to the subsequent assignment message.

4.3.4 Let's keep in touch

When the topology changes, a node might find a parent with a shorter route towards the root node. In such a case, the node should advice the previous parent, in order to let him removing the interested routing entries. In this way it can update its routing table in the same time its child breaks the link, saving at least $\frac{2}{3} * \delta$ seconds, it would have waited before to know it. If the node that changes parent is assigned for a role, the algorithm takes $\xi * (hops_{A->B} + 2 * hops_{B->C})$ to assign the missed role again, where A is the "jilted" node, B is the node which has a matching routing entry and C is the node assigned for the missed role.

4.3.5 Code Changes

```
1  void Recv(message mes){
       if(mes.type=TreeMsg){
3          if(mes.root > root_){
               root_←mes.root;
5              parent_←mes.sender;
               distance_←mes.distance;
7              Timer(timeout);
               Send(TreeMsg);
9          }
           else if(mes.root=root_ && mes.distance < distance_){
11             parent_←mes.sender;
               distance_←mes.distance;
13             Timer(timeout);
               Send(TreeMsg);
15         }
           else if(mes.root=root_ && parent_=mes.sender){
17             OnHeartbeat();
               Timer(timeout);
19             distance_←mes.distance;
           }
21     }
       else if(mes.root=root_ && (mes.receiver=here_ ||
23             (mes.receiver = CHILDREN && mes.sender=parent_) ) ){
           OnProcessMessage(mes);
25     }
   }
```

Listing 4.5: Receive

The algorithm starts with the **Recv()** function (list. 4.5). This version is pretty different from the lifetime one. In particular it analyzes the message arrived in every case, while in the lifetime version the *count to infinity problem* could block it before processing. The timer of normal nodes is set to *timeout* instead of *timeout*/3 seconds. This is because the timer is used only to reset the node in the case that the link to the parent broke, and not as clock like in there. Finally, if an heartbeat is received (cf. lines 16-19), the *OnHeartbeat()* function is called. Actually in the lifetime version such function was called by the timer running out in the *TimeOut()* function. Such version of the *Recv()* function is the same of the original algorithm, given that the spanning tree algorithm is not changed.

The **OnProcessMessage()** function (listing 4.6), is different from the lifetime version as well. The first point consists in the actions opened by a *SubMsg*

```
void OnProcessMessage(message mes){
28    if(mes.type="SubMsg"){
          if((mes.type,mes.sender)∈(S_here ⋃ S_RCV)){
30            RefreshSubscription(mes.type,mes.sender);
          }
32        else{
              InsertNewSubscription(mes.type,mes.sender);
34            Send(SubMsg,mes.flag);
          }
36    }
      else if(mes.type="PubMsg"){
38        if(DecodeFlag(mes.flag)="DataMsg")
              ( ... )
44        else if(DecodeFlag(mes.flag)="SingleMsg")
              ( ... )
53        else if(DecodeFlag(mes.flag)="Remove")
              ( ... )
60        else if(DecodeFlag(mes.flag)="Lets"){
              PSType* ris←Remove(mes.sender);
62            int lenght←how_long(ris);
              for(int i← 0; i < lenght; i++){
64                if(Is_a_role(ris[i])=1)
                      if(Reassign(ris[i]=1)
66                        continue;
                  if(ris[i] ⋂ (S_RCV ⋃ S_here)≡∅)
68                    Send(PubMsg,ris[i],EncodeFlag("TypeLost"));
              }
70        }
          else if(DecodeFlag(mes.flag)="TypeLost"){
72            Remove(mes.sender,mes.type);
              if(Is_a_role(ris[i])=1)
74                if(Reassign(ris[i]=1);
                      return;
76            if(mes.type ⋂ (S_RCV ⋃ S_here)≡∅)
                  Send(PubMsg,mes.type,EncodeFlag("TypeLost"));
78        }
      }
80 }
```

Listing 4.6: On Process Message

reception. If the arrived subscription is a new one the message is forwarded at once, accordingly to the fast notification technique. Moreover, a *PubMsg* takes in

this version more different meanings. In particular two more flags could be encrypted in a *PubMsg*, one to implement the *fault notification*, the other one for the *LetsKeepInTouch* functionality. In the case the published message has the *"Lets"* flag, the node removes all the sender subscriptions by the *Remove()* function. Such function returns a list of all the removed types and, for each of them, if any other child is subscribed for the same type, and the node itself as well, a *PubMsg* with the flag *"TypeLost"* is sent to the parent (cf lines 67,68). Otherwise, if the type lost is a role, and the node has a checking availability among the children or its ones, it reassign the role (cf. lines 63-65). Such operation is carried out by the *Reassign()* function, which takes in entry a *PSType* of a role and tries to reassign it. If it succeeds it returns *1*, otherwise *0*.

The other new flag introduced by this version is the *TypeLost* one. If a published message with such a flag is received, a child has expressly removed the subscription for a particular type, so the *Remove()* function is called (cf. line 72). Then, if the removed subscription was a role activation, the algorithm tries to reassign that role. If it does not succeed, or the type was not a role and the node and its children are not subscribed for that type, the message is forwarded to the parent (cf. lines 76,77).

```
void Time_out(){
82    if(here_≠ root_){
        Remove_Roles_Activations();
84      for(∀(x,m) | r_xm ∈ R Λ r_xm ∈ P_RCV)
          Send(PubMsg, EncodeType(r_xm|"Active"),
86                                EncodeFlag("Remove")));
        recv_←0;
88      root_←here_;
        parent_←here;
90      distance_←0;
        Send(TreeMsg);
92      Timer(timeout/3);
    }
94  else if (here_=root){
        Send(TreeMsg);
96      OnHeartbeat();
        Timer(timeout/3);
98  }
}
```

Listing 4.7: Time Out

```
100 void OnHeartbeat(){
    Age_Routing_Table();
102 PSType* ris← Remove_Old_Entries();
    RoleAssignment(ris);
104 if(here_≠ root_)
        Send(SubMsg);
106 }

108 void RoleAssignment(PSType* lost){
    for(∀(x,m) | r_xm ∈ R){
110     ( ... )

115 }
    if(here_≠ root_){
117     int lenght←how_long(lost);
        for(int i←0;i < lenght; i++){
119         if(Is_a_role(lost[i])=1)
                if(Reassign(lost[i])=1)
121                 continue;
            if(mes.type ⋂ (S_RCV ⋃ S_here)≡∅)
123             Send(PubMsg,mes.type,EncodeFlag("TypeLost"));
        }
125 }
    else if(here_=root_){
127     ( ... )

133 }
}
```

Listing 4.8: OnHeartbeat and Role Assignment

The function **TimeOut()** (listing 4.7), changes its functionality in a significant way. In the lifetime version it was used like a clock for the most part of the algorithm features. Actually, the *OnHeartbeat()* function was called by here, and heartbeats and subscriptions were sent by this function. In this version such function is called by a normal node only when the link to the parent results broken, and by the root at the timer running out (this point is the same). Accordingly with the double activation problem when a node resets itself it removes the roles activations for which it was assigned (cf. line 83), telling to children to make the same (cf. lines 85-86).

The **OnHeartbeat()** function (listing 4.8), is quite the same with respect to

```
int Reassign(PSType tipo){
136   PSType app = TypeConversion(tipo);
      if(app ∈ S_here){
138       OnActivateRole(app);
          return 1;
140   }
      else if(app ∈ S_RCV){
142       Send(PubMsg, app, EncodeFlag("SingleMsg"));
          return 1;
144   }
      return 0;
146 }
```

Listing 4.9: Reassign

the lifetime version. The only difference is that it passes to the *RoleAssignment()* function a list of the removed types. Basing on such a list the **RoleAssignment()** function tries to reassign possible missed roles activation (cf. lines 116-120), if this is not possible a message with the flag *TypeLost* is sent to the parent. The notification is sent also for "normal" missed subscriptions. The other parts of this function are exactly the same with respect to the lifetime version.

The **Reassign()** function is a new function, it provides to try to reassign roles when a missing one is detected. The operations carried out by such a function are to check if the node itself is able to take the missed role (cf. lines 136-138), or if a matching entry exists in the routing table (cf. lines 140-142). If it succeeds to reassign the role returns 1, otherwise 0.

4.4 Synthesis

In this section a unique algorithm version is proposed. Basing on the analysis made in the previous ones the better trade-off between stabilization time and energy consumption versions is pointed out. They have been realized extremely optimized for their aims, therefore they represent the two edges in the middle of which final version's issues can sweep. Besides such a trade-off, the final version proposes optimizations also focusing on the network heterogeneity.

4.4.1 Drawing

Analyzing the proposed optimizations, it is possible to see that they does not exclude each other. Let us now consider the possibility to compose them. Actually, heartbeats decoupling and message piggybacking increase the stabilization time, however not in a considerable way if coupled with fast notifications methods. On the other hand the stabilization time methods are used only during a network reconfiguration, therefore even if the device lifetime is reduced, it is not in a critic way. Thus, taken into consideration the base version the two algorithms can be merged together. Optimizations proposed for energy consumption can be adopted without big problems also in the final version, while optimizations used for the stabilization time actually affect the devices lifetime.

Besides stabilization time and energy consumption another important requirement is the network **heterogeneity**. Such a concept can be used to try to improve the energy consumption efficiency in the network, offsetting on what we lost merging the two algorithm versions.

An interesting point of directed diffusion (see sec. 2.5.2) is that, when nodes set up gradients they tune the gradients refreshing time interval by their own capabilities. Afterwards, the more efficient routes are preferred to the other ones. Hence, the system exploits powerful nodes, saving on the weak ones energy. In our system nodes closer to the root node are more loaded with respect to the other ones, and given that powerful nodes would exist in the network, they could be used to carry out an higher traffic rate.

Through the concept of heterogeneity of power capacity, it is possible to reorganize also the rules to assign roles to capable nodes, in order to prefer the powerful ones. The results of such operations would lead to concentrate computation on a limited number of subtrees, composed by an high percentage of powerful nodes. Nodes belonging to the other subtrees could slow down their heartbeat periods, thus saving even more energy with respect to the other ones.

4.4.2 Classes

In a heterogeneous network not all the nodes have the same energy capabilities. From this point of view it is sensible to load mostly on powerful nodes, trying to reduce the energy consumption of weak nodes. Therefore, we can now think to change the algorithm in order to consider two devices classes, one powerful and one weak, and to exploit mostly powerful ones. Among the algorithms composing the stack the spanning tree is the one which more can make use of such information. In particular in figure 4.5 there is tree structure we would to achieve.

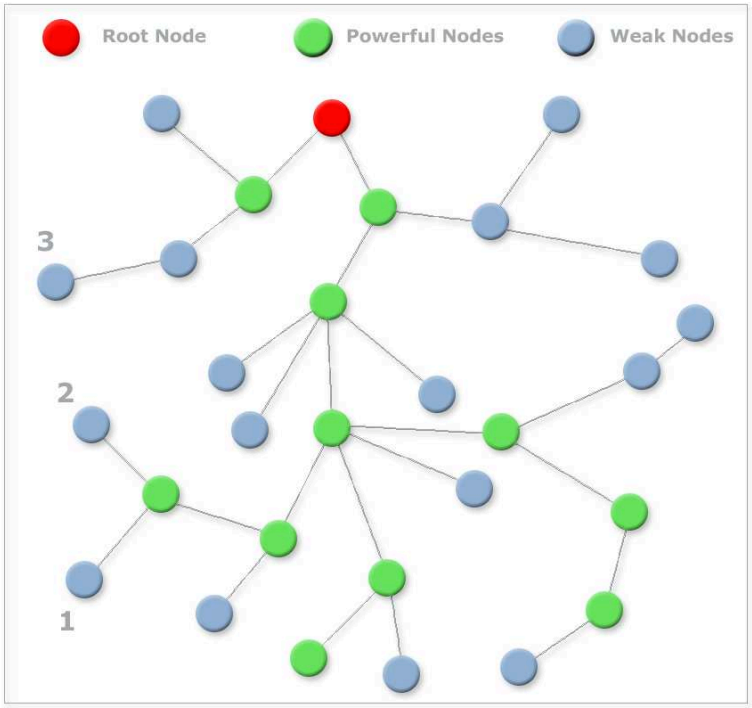

Figure 4.5: Aimed Tree Structure Format

In such a structure powerful nodes represent a *backbone* for the whole tree. Nodes should always prefer powerful nodes as parents, therefore weak nodes are always leaves, excepting when any powerful node is available in the same transmission range. Moreover, in order to make such choice more effective, nodes should be able to select the parent basing on the number of powerful nodes along the whole path to the root. To do this a new information has to be coded in messages, and available in every message exactly like the distance field.

While distances being equal, the parent with an higher powerful factor has to be preferred, but the contrary not always lead to effective tree buildings. Actually,

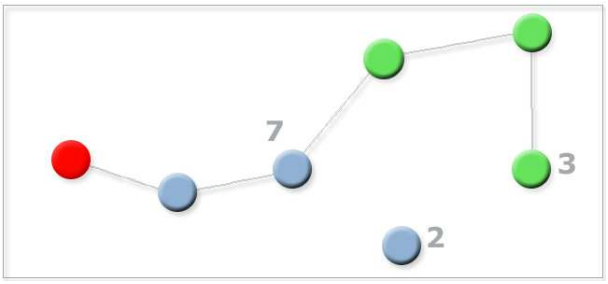

Figure 4.6: Parent Choosing

in figure 4.6 node *2* must prefer *7* as parent, even if any powerful node is available in that path. By choosing node *3* as parent, nothing would be improved, given that anyway node *7* would have new subscriptions in its routing table. Moreover, node *2* in such a way would stay at the sixth level instead of its natural third, leading the system to waste energy instead to save it.

To identify the convenience in choosing parents, we can define two parameters:

$$Gain_{distance} = distance_{new} - distance_{present} \tag{4.1}$$

$$Gain_{power} = descent_{new} - descent_{present} \tag{4.2}$$

The $Gain_{distance}$ is the parameter used up to now to choose a parent, while $Gain_{power}$ points out the parent which have a path to the root with the higher number of powerful nodes (*descent*). Actually, we can be sure that the new path is more convenient of the present one only if what we loose from the hops number point of view, is compensated by a greater number of powerful nodes. Moreover, by using such a criteria, the creation of cycles is avoided. Parent is therefore chosen so:

$$parent = ((Gain_{power} - Gain_{distance}) > 0)?new : present \tag{4.3}$$

To implement such a mechanism a new information has to be present in every message header. However any new field is needed in the header message format. Actually the decimal part of each distance field can be used to encrypt it. Nodes will add to the distance field of sent messages a term equal to the weight average between the parent's descent and *0.0* or *0.9*, depending if the node is a weak one or not. Receivers, by such information and by the hops distance, are actually able to obtain the composition of the path to the root node.

4.4.3 Assignments Policy

To reduce the number of packet transmissions needed to deliver a published message, we can consider to change the policy to assign roles. When more nodes are available for a role, the algorithm chooses one in a random way. However, if all the roles of an application are assigned to the same subtree the message does not need to bubble up till the root before to be routed to the right node. In the worst case, in which the sender and the receiver are two leaves which branches meet themselves just by the root, a message has to be routed along the whole topology.

For instance, in figure 4.5 nodes *1* and *3* have been assigned for two roles of the same application. When node *1* sends a message to node *3* it has to be transmitted nine times before to get it, besides weighing on the root node. Such worst case cannot be changed, but the average time to deliver a message can be reduced. In the example in figure 4.5 node *2* has the same availability of node *3*. By choosing it instead of *3*, a great number of transmissions are saved, as well as the root is relieved of such a traffic.

The new **policy** follows these points: if any role of the application have been assigned in the tree (subtrees), the role is assigned to the matching child with the higher number of availabilities for such application. Otherwise, the role is assigned to the matching child with the greater number of roles already assigned for that particular application.

4.4.4 Idle Status

This point takes into consideration the algorithm active status, therefore when all the roles have been successfully assigned. In this case, nodes which are not assigned for any role, and which have any child assigned for a role, might slow down the subscriptions refresh interval, and then the heartbeat period. Actually, up to the moment in which a role results missing, and then a new configuration phase is opened, a lower heartbeat period can be used. In such a situation the node is assumed to be in **idle** status.

In order to do that each node, during the active phase, must set the TTL field of entries not linked to active roles to a double value with respect to the normal one. Even if such point can increase the stabilization time, nodes not assigned for roles can save much energy during the algorithm active phase. For instance, let's assume node *2* as capable to take the role *A1*, and temporarily idle. In this moment the *2*'s parent has a subscription for the type (*A1, possible*), with a TTL set to the double of the normal value. If node *2* faults, the parent detect it, and then removes its entries, only after the double of the normal time. Stabilization time may be higher if node *2* faults right when the role *A1* results missing, and the algorithm choses just it to take it such role.

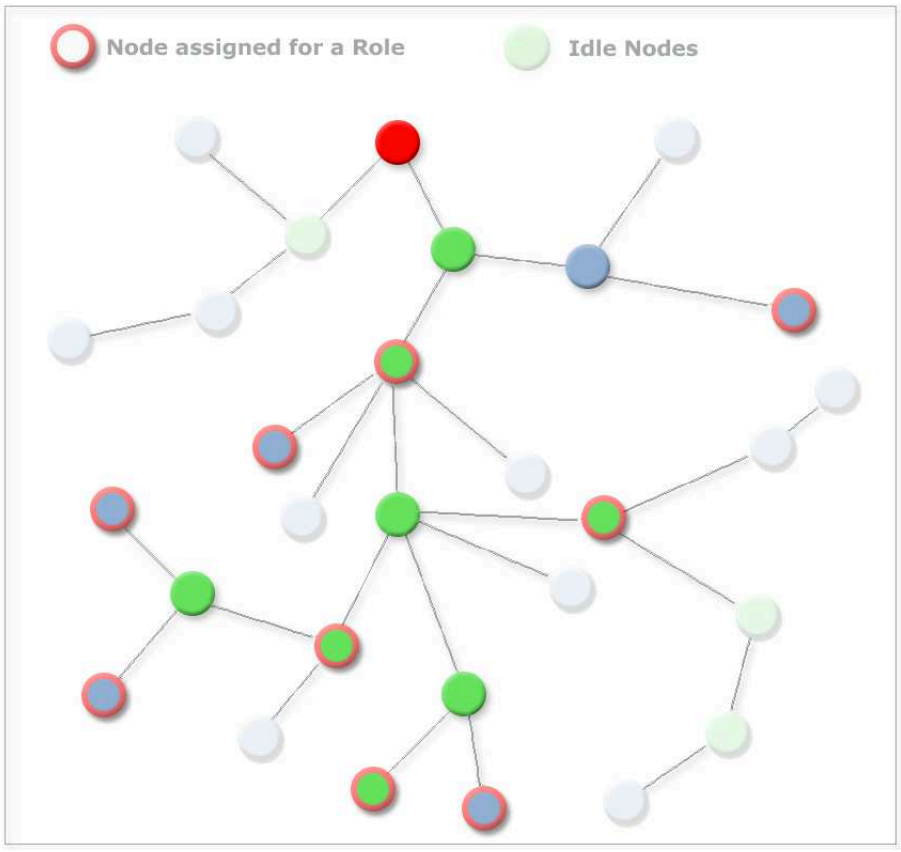

Figure 4.7: Active Algorithm Status

However, this is quite a complicate occurrence, while adopting such strategy unused nodes save during active phases about a $\frac{1}{4}$ of the energy consumption. Moreover, by using the tree structure outlined before, a weak node has got a lower probability to have a child assigned for a role. Actually weak nodes have a lower probability to have children given that powerful nodes are chosen as parent with an higher rate. Therefore, right weak nodes mostly take advantages of the idle status introduction.

In figure 4.7 the same topology of figure 4.5 is shown when the algorithm is in the active status. To encode information about the algorithm status in the message header, the distance field is used again. In fact, the sign of such value will identify if the algorithm is active (*positive*) or not (negative). To do this, from now on the root node will sends a distance equal to 0.001 (-0.001) instead of the usual 0.0.

4.4.5 Code Changes

Accordingly with the changes proposed in the previous sections, this **Recv()** function (listing 4.10) takes the lifetime one (listing 4.1) as starting point. Once solved the count to infinity problem, the activation status of the algorithm is checked (cf. line 3). Therefore the sender's descent is obtained (*numapp*) and then used to calculate the eventual gain in choosing the sender as parent (cf. line 6). The choice of the root node does not change with respect to the lifetime version, while the

```
void Recv(message mes){
2    if( Count_to_Infinity_Problem(mes.sender, mes.distance)){
        active_←(mes.distance > 0)? 1 : 0;
4       int factapp←abs(mes.distance - (int)mes.distance);
        int numapp←(abs((int)mes.distance_)*factapp)/0.9;
6       int gain←(numapp-descent_)-(abs((int)mes.distance)-distance_);
        if(mes.root > root_){
8           root_←mes.root;
            parent_←mes.sender;
10          distance_←abs((int)mes.distance);
            descent_←numapp;
12          Timer(timeout);
            Send(TreeMsg);
14      }
        else if(mes.root=root_ & gain > 0){
16          parent_←mes.sender;
            distance_←abs((int)mes.distance);
18          descent_←numapp;
            Timer(timeout);
20          Send(TreeMsg);
        }
22      else if(mes.root=root_ & parent_=mes.sender){
            recv_←0;
24          distance_←abs((int)mes.distance);
            descent_←numapp;
26      }
        if(mes.root=root_ & (mes.receiver=here_ ||
28          (mes.receiver = CHILDREN & mes.sender=parent_) ) ){
            OnProcessMessage(mes);
30      }
    }
32 }
```

Listing 4.10: Receive

parent choice takes the gain parameter instead of the simple distances information (cf. line 15).

The *OnProcessMessage()* takes the stabilization time one (listing 4.6) as starting point and does not changes it effectively, therefore it is not shown here. Listing 4.11 shows the **TimeOut()** function, it actually results as a combination of the two previous versions (listings 4.3 and 4.7). The first *if* block (cf. lines 89-97), takes into account the case in which the node is still connected to its parent, the only difference with respect to the lifetime version is represented by lines 93-95.

```
   void Time_out(){
88     Count_To_Infinity_Problem(set);
       if(root_≠ here_ && recv_< limit_){
90         recv_++;
           OnHeartbeat();
92         Send(SubMsg);
           if(active_ & (P_RCV ⋃ P_here) = ∅)
94             Timer(timeout);
           else
96             Timer(timeout/3);
       }
98     else if(recv_=limit_ && here_≠ root_){
           Count_To_Infinity_Problem(reset);
100        Remove_Roles_Activations();
           for(∀(x,m) | r_xm ∈ R Λ r_xm ∈ P_RCV)
102            Send(PubMsg, EncodeType(r_xm|"Active"),
                                   EncodeFlag("Remove"));
104        recv_←0;
           root_←here_;
106        parent_←here;
           distance_←0;
108        descent_←0;
           Send(TreeMsg);
110        Timer(timeout/3);
       }
112    else if (here_=root){
           Prepare(TreeMsg);
114        OnHeartbeat();
           Timer(timeout/3);
116    }
   }
```

Listing 4.11: Time Out

Depending on the activation algorithm status, and on the roles activation presence among the children and the node itself, timer is set to two different values. The second *if* block (cf. lines 99-110), represents the case in which the link to the parent is assumed as broken. Count to infinity problem is evaluated first, therefore role activations are removed. For each role activated by a child, a message is sent to it in order to make it removing such activation. Finally status variable are set to their default value.

The algorithm goes on with the *OnHeartbeat()*, the *RoleAssignment()* and the *Reassign()* functions. Actually, they follow the corresponding ones for the stabilization time version (listings 4.8 and 4.9), therefore they are not further shown here.

Chapter 5

Implementation

In this chapter the algorithm versions are implemented in order to evaluate the proposed optimizations To make this, in section 5.1 a simulator engine is chosen basing on the characteristics of the available ones. Such simulator is then in section 5.2 better described. Section 5.3 points out the way to implement the algorithm, while section 5.4.1 identifies a load model in order to show the functioning under different stress conditions. Metrics and issues expected by the simulations are characterized as well.

5.1 Simulator Engines

Several simulators have been developed to evaluate wireless systems. It is of primer importance to point out the one which can outline information we search in the best way. Criteria to identify it are the possibility to obtain the system's issues to which such thesis is interested in, the respective accuracy, finally the facility to obtain them. Therefore, trifling parameters like the available literature and the popularity of a simulator are taken into consideration together with the fundamental ones, like the results quality, customizability and scalability. Actually, a distinction may be done, between emulators and simulators. Following analyze bases on the work of [Cur04].

Emulators are engine which accurately reproduce the devices functioning. This means that, fixed a device, say Mica2 mote, the emulator reproduces the operations made exactly by the device's machine language. Moreover some of them lets the user specify customized hardware architectures. These systems can be used to study the detailed behavior of dedicated systems, in which applications, communication protocols and devices' hardware architecture are already known. Emulation is very useful for fine-tuning and looking at low-level results.

Simulators are engine which reproduce the system behavior leaving the internal device implementation out of consideration. Actually, simulators are extensible in order to let users to better describe particular hardware implementation. However, simulators do not use effective machine languages, therefore internal devices functioning can be studied just in a quite rough way. Simulators are properly used when looking at things from a high view. The effect of routing protocols, topology and data aggregation can be see best at a top level and would be more appropriate for simulation.

Among emulators one of the most used is *Tossim* [LLWC03]. Tossim is designed specifically for TinyOS applications, which is an operating system running on MICA Motes. Tossim owes much to its tightly coupling with TinyOS, which is one of the most used operating system for sensors. As drawback, Tossim is not extensible, or rather its extension for other platforms results very difficult [Nic06], and completely left to users. The emulator architecture is component-based. This brings a good scalability of the system, which can emulate networks with a very large nodes number. A trade-off between emulation speed and accuracy is done, however ensuring an effective level of both.

ATEMU [PBM+04] is compatible with the MICA platform as well. However, it improves the accuracy with respect to Tossim, thanks to a cycle-by-cycle strategy to run application code. This is possible because the MICA CPU functioning is emulated as well. While the emulation results more fine-grained, ATEMU sacrifices speed and scalability. In the middle between Tossim and ATEMU is placed *Arvora* [TLP05], emulator implemented in Java and characterized by an object oriented architecture. Arvora lets the user to modulate the trade-off between speed and accuracy. Such emulator is quite new, and no literature is available.

Among simulators *Network Simulator 2* (NS-2) [ns206] stands out definitely, given that on the way became quite a standard for algorithms simulation. The architecture is object oriented, and the whole system is thought to be as modular as possible. Actually, NS2 benefits of a large developers community, which, thanks to the platform extensibility, ensures new modules for coming technologies. As drawback of its extreme modularity, scalability is considerably restricted.

GloMoSim [ZBG98] is designed specifically for mobile wireless networks. Realized with an architecture object oriented and modular, unlike ns-2 GloMoSim preserves scalability thanks to a different computation organization. Actually, such is particularly optimized for parallel environment, it is written in Parsec, an extension of C for parallel programming. However just IP networks can be simulated, and this really restricts its use for sensor networks. Furthermore, last free update was released in 2000, and now it is updated as a commercial product.

OMNet++ [MSK+05] is a modular component-based simulator. To simulate sensor networks OMNet++ uses a model called SensorSim. The component-based architecture is particularly efficient and it is significantly faster than ns-2. Furthermore, it accurately models most hardware and includes the modeling of physical phenomena. Actually, very little published work has been done using SensorSim, and very few protocols have been implemented yet.

Finally, *Sidh* [Car04] is a recent component-based simulator written in Java. Specifically designed for wireless networks, its modularity is one of the best. Systems can be described in a very accurate way, and propagation and hardware models are easily exchangeable. As drawbacks, its large memory consumption restricts the scalability, moreover any literature is available yet.

Engine	**Type**	**Scalab.**	**Accur.**	**Exten.**	**Popul.**	**Facil.**
Tossim	em.	very good	good	no	good	avg
ATEMU	em.	avg.	very good	no	low	good
NS-2	sim.	low	avg.	very good	very good	good
GloMoSim	sim.	good	avg.	avg.	avg.	good
OMNet++	sim.	good	avg.	avg.	low	low
Sidh	sim.	good	avg.	avg.	low	good

Table 5.1: Engines Comparison

Our aim is to obtain good and reliable information about the algorithm working. It sets aside from particular hardware realization, and we are not interested in knowing the detailed device's internal mechanism as well. Rather, the algorithm complexity, load distribution and routing efficiency are the points by which we can validate the algorithm. Thus, emulators are not taken into consideration. Among simulators the possibility to compare the results with a large literature, to rely on a sound community of developers, drive the choice to the **network simulator 2**. Furthermore, its extensibility guarantees in future the possibility to evaluate the algorithm in different environments from the ones decided in a first moment.

5.2 Network Simulator 2

Ns-2 is an object oriented simulator, written in C++, with an OTcl interpreter as a frontend. The simulator is composed by a class hierarchy in C++, and a similar class hierarchy within the OTcl interpreter. The two hierarchies are closely related to each other. From the users point of view, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy. The root of this hierarchy is the class TclObject .

Users create new simulator objects through the interpreter; these objects are instantiated within the interpreter, and are closely mirrored by a corresponding object in the compiled hierarchy. The interpreted class hierarchy is automatically established through methods defined in the class TclClass. user instantiated objects are mirrored through methods defined in the class Tcl Object. There are other hierarchies in the C++ code and OTcl scripts; these other hierarchies are not mirrored in the manner of TclObject.

Ns-2 uses two languages because simulator has two different kinds of things it needs to do. On one hand, detailed simulations of protocols requires a systems programming language which can efficiently manipulate bytes, packet headers, and implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important.

On the other hand, a large part of network research involves s lightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important.

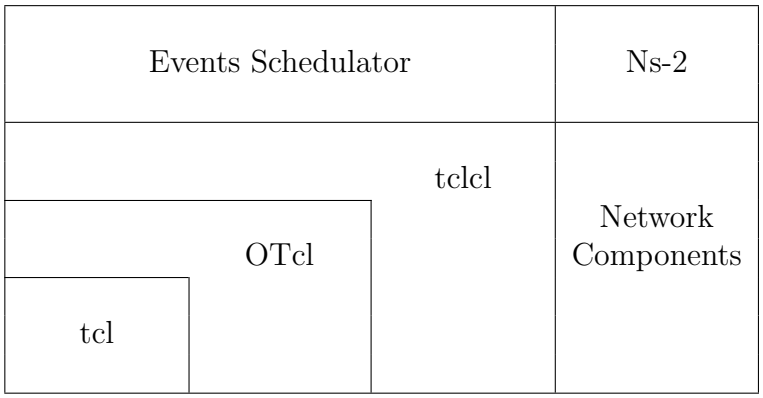

Table 5.2: Ns-2 Architecture

Ns-2 meets both of these needs with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. Ns-2 (via tclcl) provides glue to make objects and variables appear on both languages. The architecture is represented in table 5.2

For example, links are OTcl objects that assemble delay, queueing, and possibly loss modules. To use a particular configuration of them OTcl is enough. But to use a special queueing dicipline or model of loss a new C++ object is needed. Afterwards, such new module can be used through the OTcl interface. A representative tree structure of the OTcl interface is in figure 5.1. The root is the class *TclObject*, from which all the objects are derived. Most notable is the class *NsObject*, and its subclass *Connector*, which are the base for all the networks components. Among these *Delay* and *Queue* model the link behavior, while *Agent*

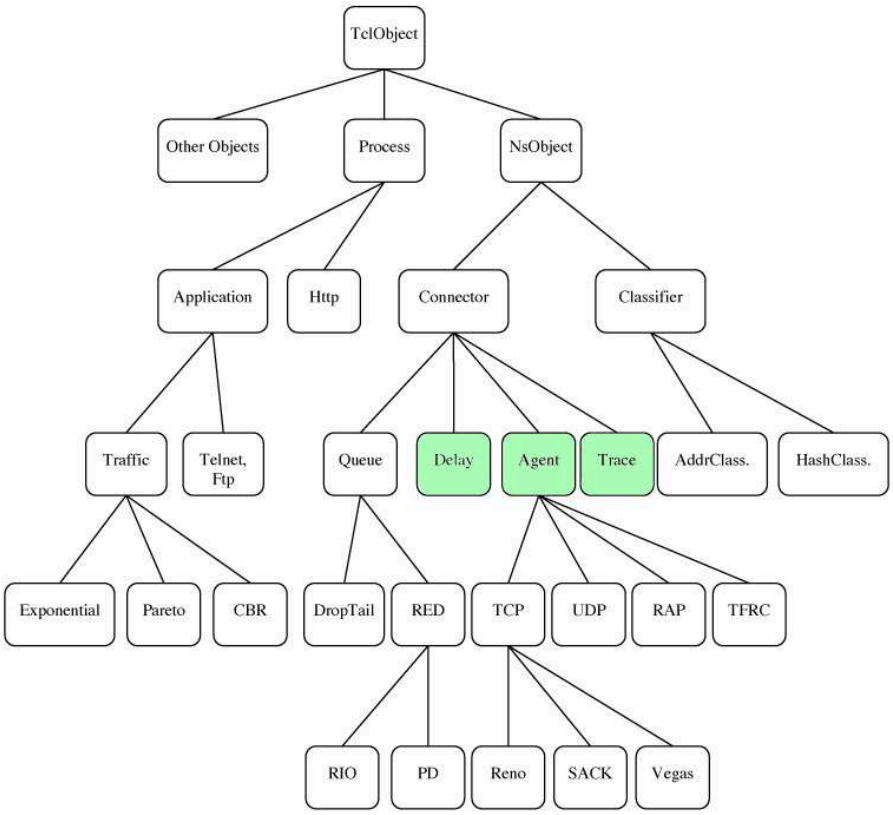

Figure 5.1: OTcl Interface Structure. In evidence classes that have been changed to implement the algorithms

is the base for all the protocols.

5.3 Simulator Implementation

To evaluate the algorithm we need to implement it as a C++ module. Such module consists of a collection of classes, among which the main one has to be derived from the agent class. In this way the module inherits all the properties needed to represent a ns-2 object. Such class features links to the simulation interface; in particular it can check values of a simulation parameter, and can be called by the simulation interface in order to trigger particular actions. The agent is linked to the underlying and overlying protocol layers as well. Such links are not fixed by the agent itself, actually they are declared and bound through the configuration interface. To implement the agent the procedure addressed by [ns2] is followed. Moreover, a new packet format has to be declared as well.

About the architecture, we choose to use a **802.11** wireless channel to simulate. This choice is due to the opportunity of future comparison, even if other wireless technologies are recently developed right for sensor networks. Actually, our goal (see sec. 1.2) is to propose an algorithm effective for the most part of radio implementation. For this reason, 802.11 represents the more sensible choice. However, from this point of view ns-2 shows a lack of customization. In fact, the 802.11 module is tightly coupled with the IP stack, hindering its use with others overlying protocols. To avoid such a difficulty, our agent is placed above an IP layer modified in the way to do not affect our algorithm behavior. A problem concerning with 802.11 is the lack of preset broadcast collision avoidance mechanism. To limit such a problem, the agent models a rough delay profile in packet transmitting.

As already said in the previous section, ns-2 is as a simulator with a low scalability. Therefore, in order to use a nodes number larger as possible, all the operations able to expedite **simulator computation** have to be adopted. Actually, operations which slows down a simulation are the object binding and tracing. About object binding all the needed operation are compacted and made at the beginning of the simulation. In this way time spent to carry out them is minimized. About the tracing, it is fundamental to analyze system behavior. The point which really slows down simulations is the continuous memory writing. Using ns-2 with default trace mechanism means that every node writes all the network events, like packets transmission and reception, and node faults. Furthermore, collisions are traced at the MAC layer, while packets can be analyzed only at overlying layers. This means that to take into consideration all the wanted issues, both MAC and Agent layer have to be traced, bringing a node to write twice for a single packet transmitted.

These considerations make evident the need of data aggregation before tracing. Two types of trace data may be distinguished, the first addresses issues affected by general network properties. Traffic profile, nodes' energy and memory consumption belong to such a category. Actually, they does not concern with particular instants of the simulation, therefore they can be traced just at the end of the simulation. The other category represents data which meanings is tightly linked with temporal events. For them the solution is to use a tracing format which can minimize the memory writings number.

5.4 Simulation Setup

In order to best evaluate the algorithm, a proper analyze of parameters which affect its behavior is needed. Two topics may be distinguished, the first deals with the network topology, that is the number of devices and their distribution. The way by which topology changes the algorithm efficiency is argued in section5.4.1. The second topic refers to the stressing conditions that may change the algorithm behavior and is analyzed in section 5.4.2.

5.4.1 Topology

The topology determines the number and the position of nodes along the area, effectively changes the behavior of the whole system. By increasing the **number of nodes** the entity of memory and energy consumption grows as well. Actually the former because more availabilities are in the network and so nodes closer to root node will have more subscriptions definitely. The latter because each node has more neighbor, and so it will receive more packets. Furthermore, collision probability grows up, making the system generally weaker. The chosen interval of nodes numbers is from 100 up to 500. The steps will be of 10 nodes, in order to have a fine-grained study.

Also the **position of nodes** significantly affects the network behavior. Fixed to 2 kilometers per 2 kilometers the simulation area, and using a transmitting range of 250 meters, the existence of nodes at the extreme position of the area is assumed. Moreover, topologies are planned in the way that all the nodes are in the transmission range of someone else. Assuming that availabilities are equally distributed and fixed the root node, three main topology types may be distinguished. An uniform nodes distribution, a distribution more dense around the root node, and the last one in which root node is at the quite at the opposite of the main part of nodes.

With a **uniform** nodes distribution (fig. 5.2 a) the spanning tree builds a tree with the minimum number of levels. This means that leaves have a minimum distance in hops with respect to every other distribution. Furthermore, each node has the same number of nodes in its transmission range and so quite an equal number of children. In such a condition, publish/subscribe creates routing tables with a similar number of entries. Hence the memory consumption is equally spread over the network, and depends almost only on the distance in hops from the root node. Just from a probabilistic point of view this means that, fixed a level (all the nodes with same distance in hops to the root) roles are assigned in a uniform way as well. The consequence of this is that data traffic, accordingly with the other traffic types, in the same level will result equally distributed. Uniform distribution of nodes represents the best topology, given that all the algorithm's components never assume pick values.

Unbalanced distribution with the root node **far** from the most part of other nodes (fig. 5.2 b) is characterized by a greater stress on nodes closer to the root. Actually, starting from the area in which nodes are more dense and going towards the root, the number of nodes decreases and the routing tables of them grows up to the same order of the root's one. Collision probability is lower around the root and much greater on the opposite side.

Unbalanced distribution with the most part of nodes **close** to the root (fig. 5.2 c) node, is characterized by a very high density. Actually the most part of the network is grouped in a small area, therefore subscriptions are very good distributed, and routing tables never reach the root's one order. However, collision probability is high, moreover around the root as well. This means that all the network activities are affected by a further weakness.

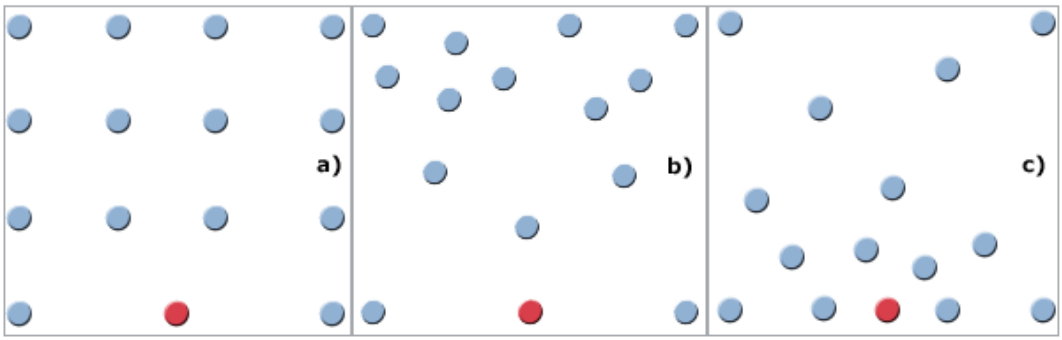

Figure 5.2: Topologies

5.4.2 Load Model

As load model we mean all the parameters which can characterize the stress of the network traffic. Apart from topology other issues which can contribute are the number of roles and faults, and the publishing rate as well. The number of **faults** stresses the network forcing the system to reconfigure. In order to study the effect of redundancy (the progressive higher number of nodes), such term is fixed. At the beginning it represents almost an half of the total number of nodes (45), up to the last step, in which it will represent just $\frac{1}{10}$ of the total nodes.

The number of **roles** affects the system efficiency as well. More roles means a greater stabilization time and frequenter unstable phases. Moreover, an higher number of published messages will be routed through the network. To study the system reaction to different entities of such a term it will assume values of 20, 40, 60, 80 and 100 roles. About roles, the number of nodes capable to take the same role is important as well. In order to better study the effects of redundancy on the algorithm ,such field is linked to the number of nodes. In a gradually way it starts from 2 in correspondence of 100 nodes, up to 6 nodes when 500 nodes are present in the network. The capabilities are distributed in a way that in every time an available node for a role is always present. This means that for each role at a least a node capable to perform it cannot fault ever.

The last issue is the **publishing rate**. In order to effectively model the load effect each node assigned for a role will publish a data message with a certain rate. Actually, a constant bandwidth model is chosen. Given that heartbeats represent a metric for the most part of system performances, it is sensible to use it in this case too. Nodes assigned for a role will publish data message every heartbeat, every two heartbeats, finally a more relaxing one, with just a message every five heartbeats.

Parameter	Values	Step
Number of Nodes	from 100 to 500	10
Topology	3 types	-
Number of Faults	45 faults	-
Number of Roles	from 20 to 100	20
Availabilities for Role	gradually from 2 to 6	-
Publishing Rate	every 1,2 or 5 heartbeats	-

Table 5.3: Simulation Parameters

Chapter 6

Evaluation

This chapter presents the simulations' result for the three optimized algorithm versions, the *stabilization time* version, the *devices' lifetime* and the *finale* ones. They are compared with the *base* version, which results to be the implementation of [WPJM06]. For each issue of interest the algorithm versions are compared, and their behavior analyzed. Section 6.1 gives an overview of the issues taken into consideration, and the analyze method. Based on them, sections 6.2 and 6.4 shows the results for the stabilization time and the energy consumption. While, section 6.3 makes in relation different load shapes with the algorithm behaviors, and section 6.5 shows the memory consumption for different working conditions.

6.1 Simulations Overview

Simulations' goals are to study the algorithms' behavior, in order to understand the efficiency of the optimizations proposed in the chapter 4. The first two algorithms have been optimized respectively for the time to stabilize (TTS) and the devices' lifetime (*Lifetime*). Consequentially, a unique version (*Finale*) composes the two aspects proposing further optimizations by taking care of the network heterogeneity. Optimized algorithms are compared with the *Base* version, which represents the starting point of such work. TTS and *Lifetime* represent the two edges in the middle of which *Finale*'s issues are expected to sweep. Therefore, first stabilization time and energy consumption, that are issues object of trade-off, are analyzed. Afterwards, the behavior of the algorithms under different working conditions is studied. Finally, the memory consumption confirms the degree of scalability of the system. In order to better identify the different components forming results, each algorithm has been simulated activating one layer at a time. Thus, first with only spanning tree running, then with also publish/subscribe activated, finally the whole stack.

6.2 Stabilization Time

Stabilization time is the time the algorithm spends to recover from a transient fault. As explained in 5.4.2 45 faults are induced during the simulation. The stabilization time might be assumed as composed by three components, one for each algorithm of the stack. In the following subsections the spanning tree of the three algorithm versions are compared.

6.2.1 Spanning Tree

In figure 6.1 the time the three algorithm versions spend to structure the spanning tree are compared. The spanning tree algorithm, that represents the base of

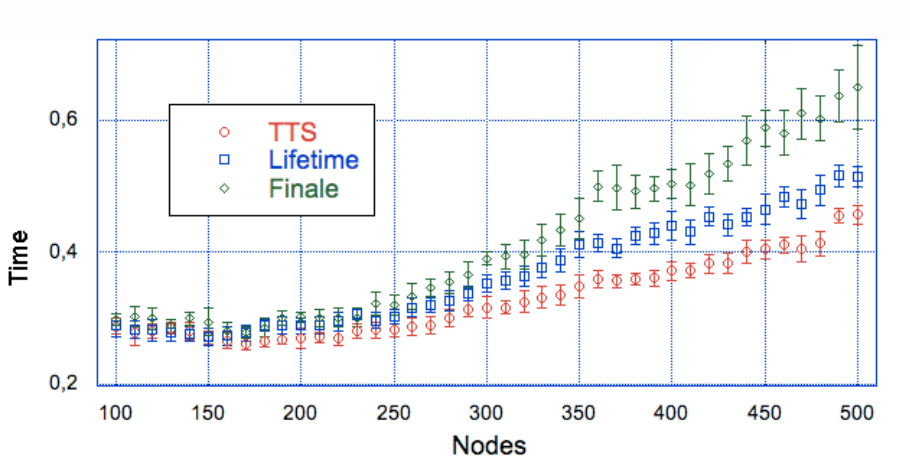

Figure 6.1: Spanning Tree Stabilization Time

the stack, stabilizes in a very short time span, and it does not actually affect the overlying algorithm stabilization time. The time is measured in heartbeat periods ($\frac{\delta}{3}$), and the 10% of nodes are assumed as powerful ones. The *TTS* version, which spanning tree algorithm is the same with respect to the base one, stabilize in the shortest time. *Lifetime* version is slower, actually it pays the delay introduced to face the *count-to-infinity* problem, as well as the heartbeat decouple. *Finale* version's spanning tree has the lifetime one as starting point, therefore its stabilization time as well. Moreover, a component due to the more laborious parent's choose is added. In fact, while the other versions just choose the shortest path to the root, *finale* takes care of the highest number of powerful nodes along the path as well.

6.2.2 Publish/Subscribe

For publish/subscribe the stabilization time represents the time span the algorithm spends to update nodes' routing tables, in the way that each of them correctly contains children subscriptions. In response to a missing subscription, the minimum stabilization time is of two heartbeat periods, that is the time by which routing table entries' TTL is set. In figure 6.2 the three publish/subscribe versions are simulated. The time is expressed again in heartbeat periods, and the number of *PSTypes* is assumed as 40, while powerful nodes are considered as the 10% . *Lifetime*, which publish/subscribe version is the same of the base one, is consider-

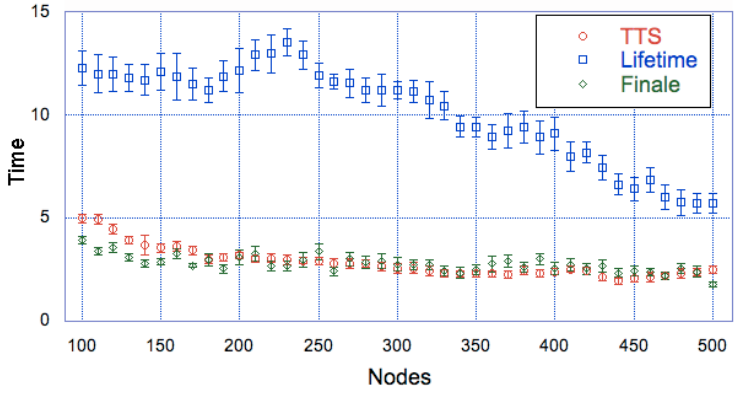

Figure 6.2: Publish/Subscribe Average Stabilization Time

ably slower with respect to the other ones. *Finale* and *TTS* use fast notifications, accelerating in this way the routing tables update, while *lifetime* not. In lifetime version when a routing entry is erased, such information spends two heartbeats periods for each layer before to reach the root node. Actually, in a denser topology (high nodes numbers) such issue decreases. This happens because with more nodes it is more possible to find a shorter path to the root, thus less hops to the root node. In lifetime version this leads to a considerable difference, given its dependency from the distance to the root node. In the same way also the other two algorithm versions reduce their publish/subscribe stabilization time, even if by a less important rate. In fact their stabilization times are already close to the minimum possible (2 heartbeat periods), and just the small component due to the message propagation is reduced.

6.2.3 Role Assignment

For the role assignment algorithm the stabilization time represents the time the algorithm spends to reassign a role, once it The simulations of the role assignment

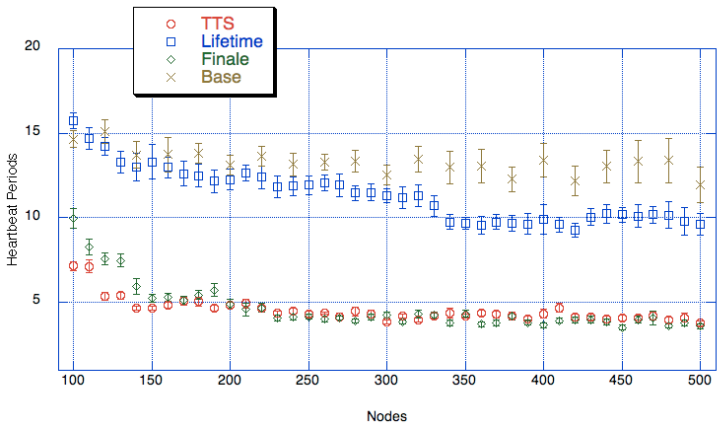

Figure 6.3: Role Assignment Average Stabilization Time

algorithm stabilization time (fig. 6.3) synthesizes issues pointed out by the publish/subscribe . Actually, a role missing is a lost subscription. With respect to the time needed by the publish/subscribe to update the routing tables, the role assignment spends an higher time to reassign the role. Among the three optimized algorithm versions, lifetime confirms its slowness, in fact the role assignment stabilization time is more than the double with respect to the other two. *Finale*, together with *TTS*, assumes the best possible result and, aside from lower nodes' numbers, its stabilization time is about a third of the base version one. Moreover, the higher nodes number does not reduce such term, on the contrary the algorithm exploits it to perform faster stabilization phases.

6.3 Traffic Effect

In this section the efficiency of the algorithm in delivering data messages is taken into consideration. Assuming 40 roles and a publishing rate of one data message every heartbeat period (the more stressing case), the behavior of the algorithms are studied. To deliver a data message the packet has to bubble up along the edges of the tree, up to the node that has the checking subscription, thus that has the receiver in its subtree. Closer to the root it will be the node by which the sender's and the receiver's subtrees cross each other, and higher it will result the number of transmissions to deliver the message. In figure 6.4 the maximum number of data messages transmitted by the publish/subscribe algorithm is shown. Thanks to the different assignment policy, *Finale* optimizes such issue, such as for higher nodes' numbers it is almost an half with respect to the other two optimized

versions. Actually, in the *finale* version nodes, when possible, assign roles of the same application to the same subtree. In this way to deliver a message an inferior number of transmission is needed.

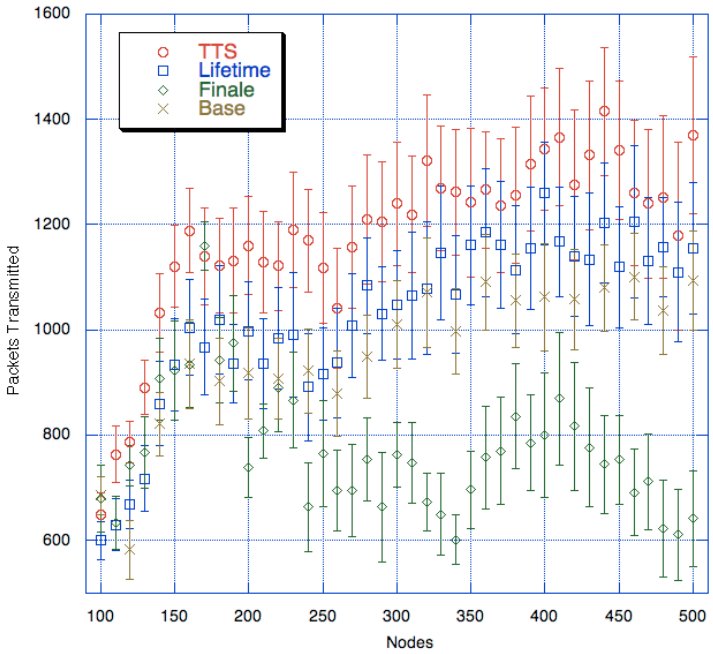

Figure 6.4: Publish/Subscribe Exchanged Data

An higher number of transmissions increases in a linear way the number of collisions as well, as shown in figure 6.5. In fact, the *finale*'s number of collisions results an half with respect to the *Lifetime* version and less than a fourth of the *Base* and *TTS* ones.

Such results lead the finale's probability of successful transmission much more higher, as the message error rate in figure 6.6 shows. In such a graph, the number of missed data messages is represented in percentage, with a roles' number of 40. The final version improves such issues as long as the nodes' number grows, exploiting in this way the redundancy of availabilities in the network.

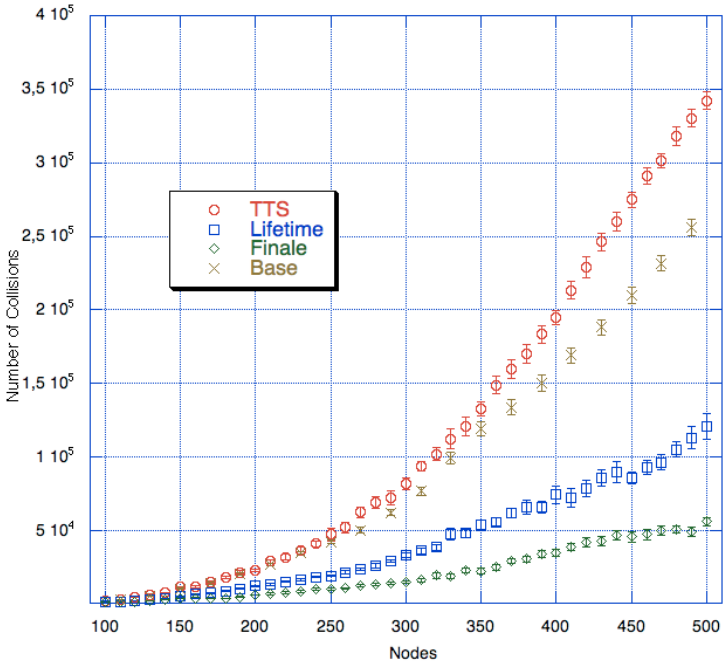

Figure 6.5: Collisions Number

6.4 Energy Consumption

Nodes energy consumption is substantially determined by the transmitted messages, and in part by the received messages as well. Analyzing our algorithm we can identify three components due to the three layers of the algorithm. The first one is correlated with the spanning tree , and in particular with the heartbeat messages. The second part belongs to the publish/subscribe mechanism, therefore to the number of subscribe messages. Finally, the third component represents the messages transmission due to the role assignment functioning and data exchanged.

Given that energy consumption strongly depends from several factors, among which the distance to the root it is not sensible to use the average values. Instead, the maximum energy consumption is adopted in order to study the worst case possible in the network.

In figure 6.7 the first component is shown. It has been obtained by simulating the spanning tree algorithm alone. Results shows how such issues is quite the same for all the algorithm's versions. Just for high nodes' numbers *Finale* is characterized by a little higher energy consumption. Actually it is due to an higher number of reconfigurations, due to a more complex choice of the parent. For the same reason *Lifetime* in the last steps consumes slightly more with respect

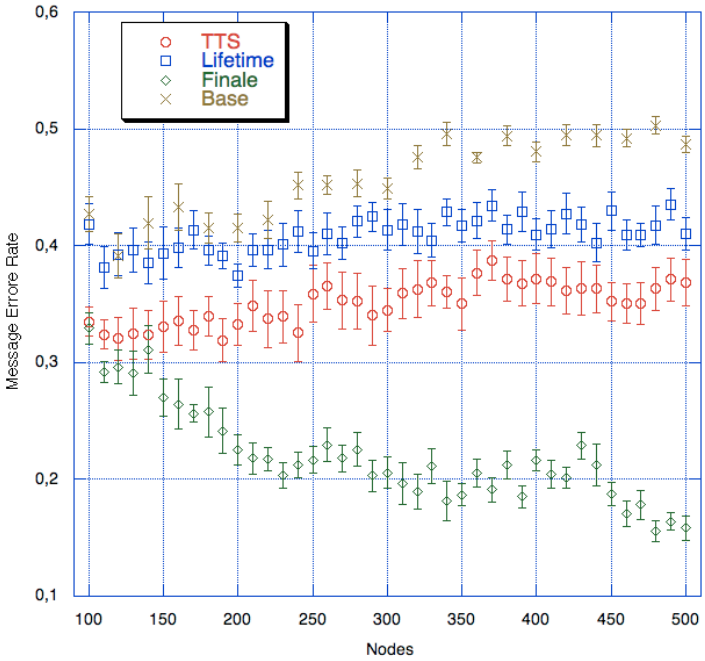

Figure 6.6: Message Error Rate

to the *TTS* spanning tree version, which has not to front the count-to-infinity problem.

The second component belongs to the publish/subscribe mechanism. In figure 6.8 the algorithm is simulated with the role assignment disabled. Therefore the shown energy consumption is due to exchanged messages for heartbeat and subscriptions. *Lifetime* spends less than the other ones. This thanks to the heartbeats piggybacking, that permits to consume just a few more with respect to the spanning tree algorithm consumption. *TTS* spends the triple of the *Lifetime* version. Here subscription messages and heartbeats are sent separately, moreover fast notifications methods provides to send an higher number of messages when a reconfiguration happens. In the middle there is *Finale*. Also it performs fast notifications, however combined with heartbeats piggybacking.

In figure 6.9 the dependence between number of roles and energy consumption is shown for the *Finale*'s whole role assignment algorithm. Actually, the load model assumed for such simulations considers that each node assigned for a role publishes message with a certain rate. In figure 6.9 it is assumed a slow rate of a message every five heartbeat periods.

In figure 6.10 the four algorithm versions are compared with respect to the energy consumption level with a roles' number of 20. Looking at the graph, the

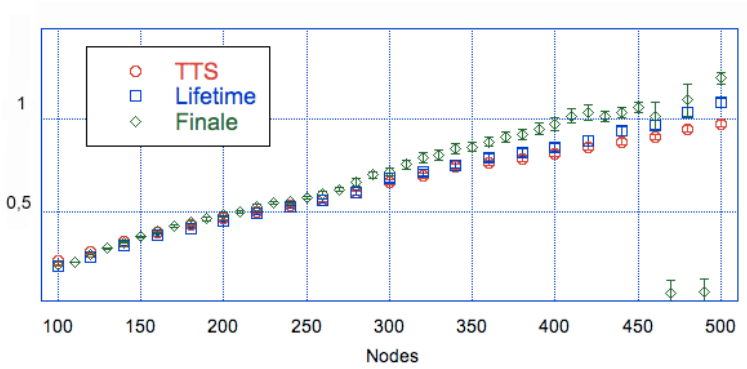

Figure 6.7: Spanning Tree Maximum Energy Consumption

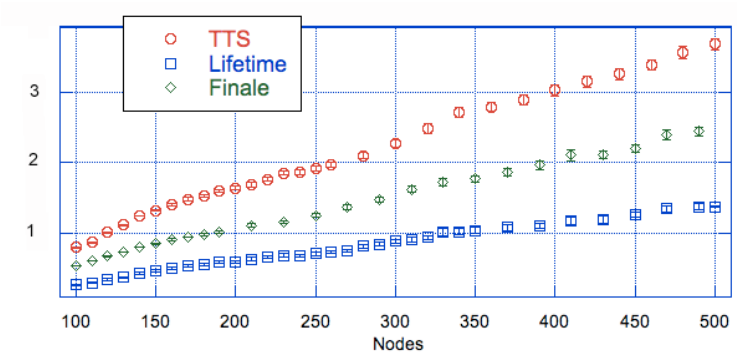

Figure 6.8: Publish/Subscribe Maximum Energy Consumption, with 40 roles

great efficiency of the final version leaps out, definitely. Not only it optimizes the energy consumption with respect to the TTS version, but, in this case and for an high nodes number, it makes better than the lifetime version as well.

This could appear in contrast with results previously shown. The reason is that optimizations featured to the role assignment algorithm make the system not only able to have a faster stabilization time, but also to save on messages transmission, thus on the energy consumption. With respect to the energy spent by the publish/subscribe algorithm (see fig. 6.8), fast notifications methods and the reassignment function allow to the *TTS* version to make up for a third of the difference with *Lifetime*. *Finale* adds to such improvements the better network structure and the clever assignment policy that brings the system to save on data messages exchanged, as previously shown in figure 6.4.

With higher numbers of roles, the number of exchanged messaged raises, and the collisions as well. However, *Finale*'s collisions shape (see fig. 6.5) does not

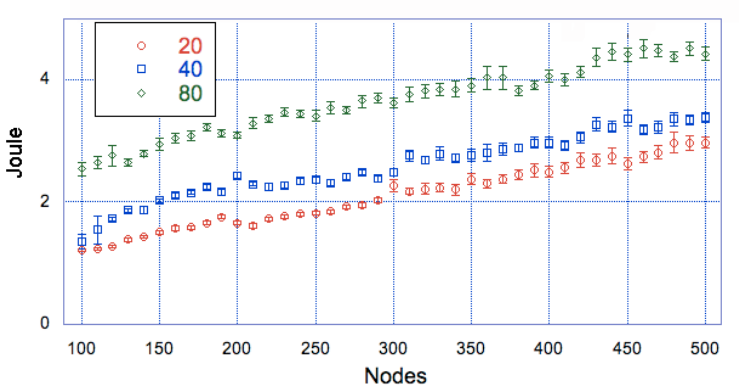

Figure 6.9: Max Energy Consumption with 20,40 and 80 Roles

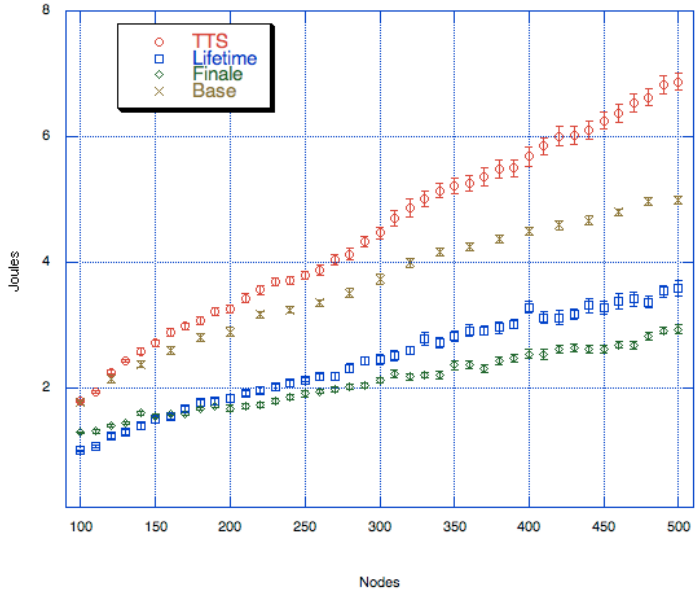

Figure 6.10: Max Energy Consumption with 20 Roles

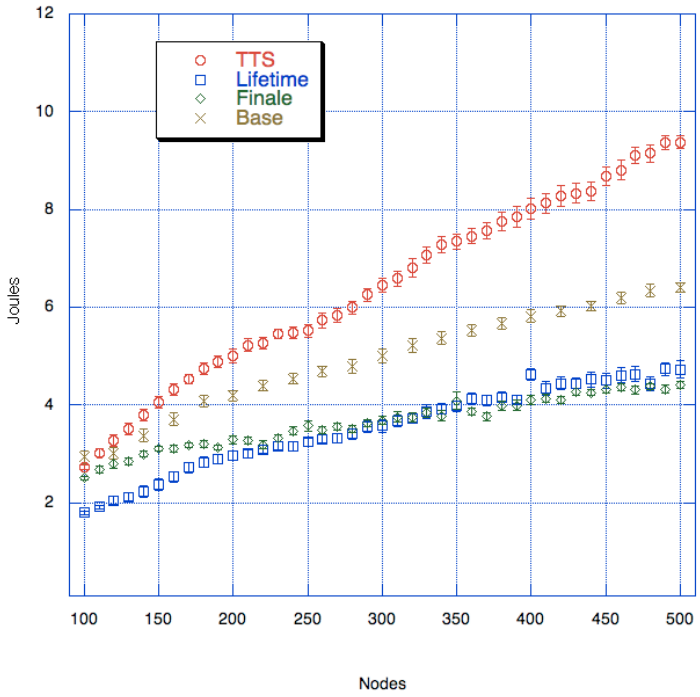

Figure 6.11: Max Energy Consumption with 80 Roles

affect in a consistent way the message delivering (see fig. 6.6), thus the energy consumption increases. On the contrary in the *Lifetime*, *TTS* and *Base* versions more exchanged messages bring to a consistent number of failed delivers, hence the energy consumption do not increase as much as it should. Therefore, for high roles' numbers, the spread at high nodes number between the final version and the *Lifetime* results quashed. In figure 6.11 the maximum energy consumption of the three algorithms, level with 80 roles, is shown. Such argue is confirmed by analyzing the averages instead of the maximum values. In figure 6.12 the average energy consumption level with 20 roles is depicted. Here, the efficiency of *Finale* version results much more stronger. Looking at figure 6.13 we can see that such gap is filled, again by the *Lifetime* inefficiency in messages delivering.

In all these graphics *Base* version assumes values between the *Lifetime* and the *TTS* ones, actually closer to *TTS*. This because it does not feature both messages piggybacking and fast notifications.

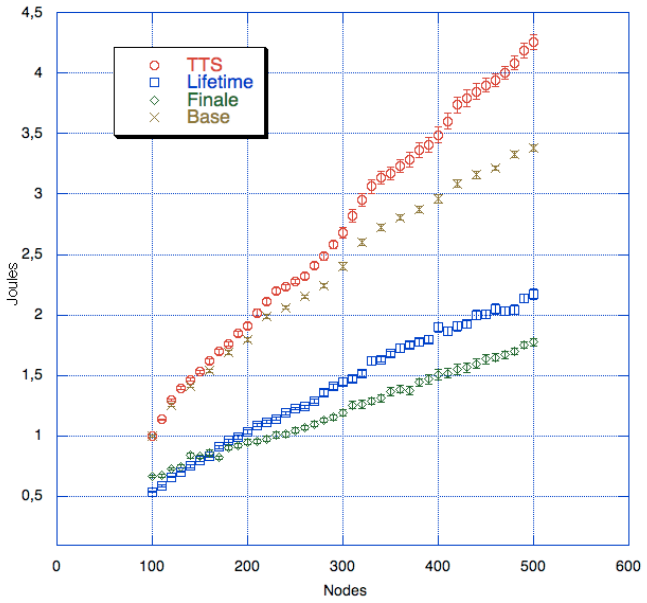

Figure 6.12: Average Energy Consumption with 20 Roles

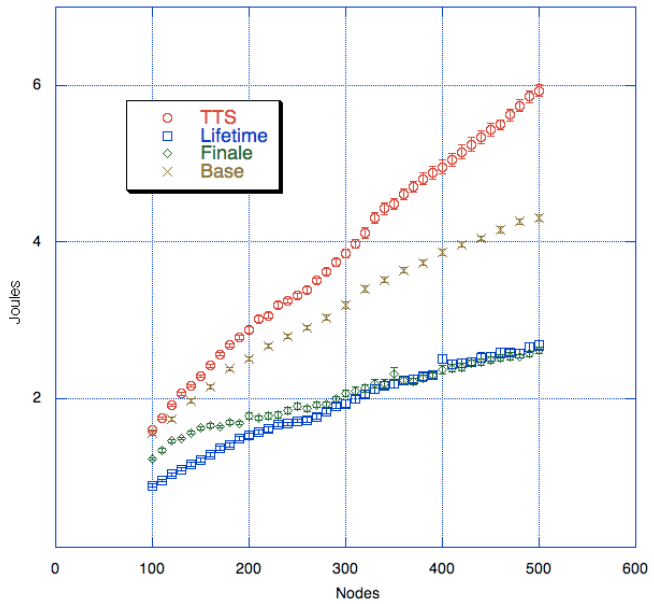

Figure 6.13: Average Energy Consumption with 80 Roles

6.5 Memory Consumption

In this work as memory consumption is intended the number of entries stored in the publish/subscribe routing tables. It represents the most important index of scalability of the system. Based on the concept of heterogeneity, *Finale* introduced the concept of powerful nodes (see section 4.4.2), through which it structure the network in backbones, over which the most part of traffic should be routed. The result of such operation leads to an higher number of routing entries into powerful nodes, in favor of weak ones. Globally, considering all the routing tables in the network, we expect a lower number of routing entries.

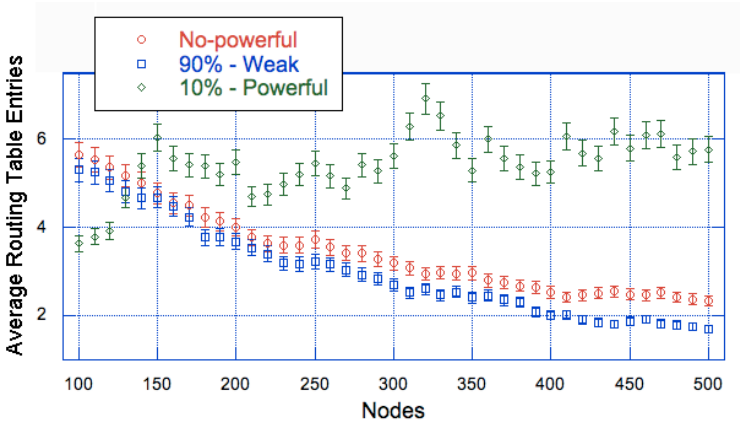

Figure 6.14: Memory Consumption with 10% of powerful nodes

In figure 6.14 the memory consumption in the *Finale* version is analyzed, in the situation in which all the nodes are weak, and with the 10% of powerful nodes. The result is that actually aggregation is made. Powerful nodes get higher routing entries numbers, thus backbones really relieve weak nodes.

In figure 6.5 the percentage of powerful nodes is brought to 30%, while in background remain the previous results. Here the situation become more consistent, and the backbone setting up appears in a plainer way. Routing tables of weak nodes are even more relieved, while the ones of the powerful nodes assume in average values close to the "no-powerful" situation. Globally, this means that the total number of routing entries in the network is reduced, and weak nodes result to have an half of the routing entries experienced in absence of backbones.

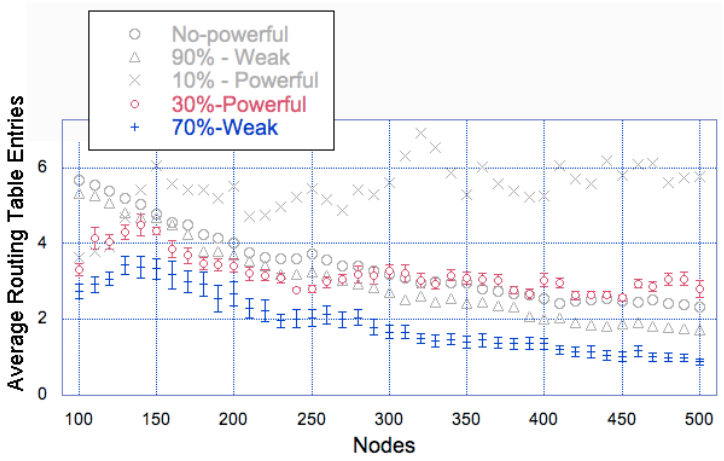

Figure 6.15: Memory Consumption with 30% of powerful nodes

Chapter 7

Conclusions

Actuator/Sensor Networks (AS-Nets) emerged in recent years thanks to research advances in highly integrated and low-power hardware. They are composed by tiny devices able to monitor physical properties of their surroundings (sensor boards), embedded controllers and computing devices, whether mobile or not. Primary goal is to monitor and control environments, based on the user's information and needs. Diffusion margins of such networks are spreading over an even higher number of applicative scenarios. Actually, a large part of an AS-Net components would be already "out there". Mobile devices equipped with even more powerful computing facilities are almost in all our pockets, sensors and embedded controllers are already entered in our houses (electrical appliances), with several aims in our cars, finally in our urban environments (surveillance cameras, electrical gates).

To manage the distributed computation in such an heterogeneous network new architectures and protocols are needed. Actually, they have to be devised in order to care of the bounded lifetime of the most part of devices, considering also the always changing network structure and the inability of users and network administrators to manage configuration phases. For these reasons self-organization and self-stabilization play in AS-Nets a crucial role, becoming fundamental for a large part of applicative scenarios.

This thesis presents a self-stabilizing role assignment algorithm, able to facilitate self-organization of the network and its application. Taking as starting point the work of Weis, Parzyiegla, Jaeger and Mühl [WPJM06], first goal was to evaluate its effectiveness. To do this related available literature has been analyzed and possible alternatives to structure the algorithm identified. Such study does not led to change the algorithm architecture but brought to several reflections that gave hints for the second goal of this thesis: the algorithm optimization.

Criteria identified as mostly affecting the AS-Nets efficiency are the *devices' lifetime*, the *stabilization time* and the *network heterogeneity*. The first one is a structural point, too small is the battery charge in the most part of devices and a

non-care of such a limit could bring to ineffective networks. The stabilization time characterizes the quality of service, it is the time the algorithm spends to adapt itself to changing environments. Finally, the network heterogeneity is indicative of the ability of the algorithm to well exploit resources available in the network. Following such criteria an improved version of the algorithm has been studied and implemented. In order to well evaluate the quality of proposed optimizations a simulator engine (NS2) has been used.

Simulations results pointed out the effectiveness of the presented algorithm for all the optimization goals. With respect to the algorithm of [WPJM06] the devices' lifetime has been improved of more than the double, and the time spent to stabilize the algorithm reaches values lower than a third. Moreover, the algorithm exploits the different power constraints of available nodes, allowing weak ones to save on consumed energy and memory space. For all these reasons the algorithm that we propose here results to better fit for the AS-Nets challenge, which is going to lead us towards a trustable and even pervasive interaction with our surroundings.

7.1 Summary

In order to better identify requirements our algorithm had to fulfill we began with a complete overview of AS-Nets. In particular, applicative scenarios and typical devices have been presented and analyzed. Afterwards, we argued the need of a distributed architecture that, in such networks, implies the implementation of a middleware layer between the operating system and the applicative layer.

Aspects of a middleware that are actually implemented by our algorithm are the communication paradigm, the fault-tolerance and the self-organization. Thus, we focused on them, analyzing available literature and pointing out several ways to build the algorithm as well. Among these choices, we found that the ones made by [WPJM06] were the most sensible for the algorithm's aims.

In particular, the algorithm resulted built in a stack fashion, composed by three layers and directly based on the radio interface. Starting from the bottom first two layers implement the communication paradigm in the form of publish/subscribe mechanism. The first one is a spanning tree algorithm and builds a totally distributed broker network in a tree fashion. Over it, the hierarchical publish/-subscribe mechanism implements needed functions to deliver subscriptions and published messages. Finally, on the top the role assignment algorithm manages to assign roles to nodes, implementing a self-organized service for the distributed computation. All the layers of the stack have been devised as self-stabilizing ones.

However, several points resulted to be actually restricting the algorithm efficiency, from the energy consumption point of view, as well as focusing on the time to perform operations.

Therefore, we identified an optimization procedure based on three optimization goals: *Stabilization Time*, *Devices' Lifetime* and *Network heterogeneity*. Given that first two resulted in conflict each other the procedure consisted in deploying two algorithms (*TTS* and *Lifetime*) extremely optimized respectively for each of the first two issues. Thus, based on analysis, the final algorithm (*Finale*) has been developed, incorporating optimizations tuned on network heterogeneity as well. In order to evaluate the quality of the proposed algorithm all the four versions (*Base*, *TTS*, *Lifetime* and *Finale*) have been implemented for a simulator engine. Among the available ones NS2 has been chosen thanks to its extensibility and its renowned reliability. With respect to the other three algorithm's versions, and for each optimization goal, simulations gained the following results:

Stabilization Time. Stabilization time has been analyzed for each of the three algorithm's layer. components belonging to the three layers of the algorithm. The spanning tree of *Finale* results to have an higher stabilization time than the other versions. This is because it structures the network in a more complex way, actually more efficient. However, the implemented optimizations in the other two algorithm's layers led *Finale* to reduce the stabilization time over than an half with respect to the base version. Actually, it achieved the same performances as *TTS*, which was developed with the only goal to speed up stabilization.

Devices' Lifetime. The algorithm's *Lifetime* version has been optimized for increasing the network's lifetime. It represents the point of reference for this optimization criteria and it is characterized by an energy consumption less than a third with respect to *TTS*, and about an half to *Base*. For small numbers of nodes *Finale* results to be in the middle between *TTS* and *Lifetime*, as one would expect. However, increasing the number of nodes, the *Finale* network structure and its different role assignment policy exploit in a clever way the nodes redundancy. In low traffic situations *Finale* succeeds to get results even better than *Lifetime* version. Even whit higher traffic situations even if *Lifetime* has the same energy consumption of *Finale* it suffers of a much more higher number of undelivered messages. Hence, normalized to the same number of delivered messages *Finale* consumes less energy per message. Such results stands out definitely, getting results considerably beyond expectations.

Network Heterogeneity. In a actuator/sensor network nodes are supposed to be not furnished with the same capabilities.*Finale* version takes into consideration the different power and memory constraints of nodes in all three algorithm layers. In particular such information are used to build up a backbone formed by stronger nodes, over which the most part of traffic is routed. In this way weaker nodes are allowed to save energy and memory space.

We can conclude that simulations results pointed out great improvements of the final version with respect to the base one, for all the chosen optimization goals. The energy consumption has been reduced of more than an half and the stabilization time results to be three time faster than the base algorithm version. Furthermore, data messages are delivered by a lower number of transmissions and with a much more higher success rate. Moreover, by taking care of the heterogeneity of the network powerful nodes are better exploited, allowing weak ones to save energy and memory space.

7.2 Outlooks

There are several aspects of the algorithm we proposed that could be deepened and extended, mostly dealing with the algorithm itself. About them we address possible ways to further improve the algorithm efficiency.

Role Placement. Final version already features an assignment policy studied to improve the efficiency of the distributed computation. In particular, without exchanging further information it tries to assign roles belonging to the same application to the same subtree. A deeper analyze could be make, by taking care of the subscribers' classes. Actually, the optimum would be to assign roles of the same application to powerful nodes close among them and connected each other through other powerful nodes. This would bring an higher computation cost, that a dedicated analyze could point out.

Routing. Hierarchical publish/subscribe is a good way to implement the communication paradigm. We argued this choice as the most sensible given the hierarchical flowing of subscriptions. However, depending on the traffic conditions other approaches could result to be more efficient. A deeper study could identify more flexible routing algorithms, able to optimize the message delivering.

Tree Structure. In our algorithm the network is structured in a tree fashion. The possibility to build not just one, but several trees could be taken into consid-

eration. In this way the load on the tree upper levels would better spread over the network, allowing weak nodes to save even more energy and memory space.

Security. Transmission security was not a goal of this thesis. However, in order to propose services for commercial use the security has to be implemented in the most affordable way. Actually, AS-Nets deal with user's more intimist information. Its position, its habits could be available to malicious listeners in every moment. We conceive AS-Nets as an instrument to easy people everyday life, and not to observe them. Further works could focus on such theme, proposing mechanisms to feature secure transmission able to guarantee users' private information.

Bibliography

[AKY91] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self stabilizing protocols for general networks. *LNCS*, 486:15–28, 1991.

[Car04] T. Carley. Sidh: A wireless sensor network simulator. *ISR Techical Reports*, 2004.

[Cur04] D. Curren. A survey of simulations in sensor networks. Technical report, University of Binghamton, 2004.

[DFFMM06] Henri Dubois-Ferriére, Laurent Fabre, Roger Meier, and Pierre Metrailler. Tinynode: a comprehensive platform for wireless sensor network applications. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 358–365, New York, NY, USA, 2006. ACM Press.

[Dij74] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 1974.

[Dol00] S. Dolev. *Self-Stabilization*. MIT Press, 2000.

[Gär03] F. C. Gärtner. A survey of self-stabilizing spanning tree construction algorithms. *Swiss Federal Institute of Technology (EPFL) School of Computer and Communication Sciences Technical Report IC/2003/38*, 2003.

[HMJ05] K. Herrmann, G. Mühl, and M. A. Jaeger. A self-organizing lookup service for dynamic ambient services. *25th International Conference on Distributed Computing Systems (ICDCS 2005), Columbus, USA*, 2005.

[IGE00] C. Intanagonwiwat, R. Govindan, and D. Estring. Directed diffusion: A scalable and robust communication paradigm for sensor networks. *MobiCom 2000, Boston, USA*, 2000.

[Kli] R. M. Kling. Mote: An enhanced sensor network node, http://www.intel.com/research/exploratory/motes.htm.

[LLWC03] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. *Proceedings ot the 1st international conference on embedded networked sensor systems*, pages 126–137, 2003.

[LLWD] P. Levis, N. Lee, M. Welsh, and D.Cullers. *TinyOS: An Operating System for Sensor Networks.*

[MFJWM05] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. Codeblue: An ad hoc sensor network infrastuscture for emergency medical care. Technical report, Harvard University, 2005.

[MHH] S. Madden, J. Hellerstein, and W. Hong. *TinyDB: In-Network Query Processing in TinyOS.*

[MJH+05] G. Mühl, M. A. Jaeger, K. Herrmann, T. Weis, L. Fiege, and A. Ulbrich. Self-stabilizing publish/subscribe systems: Algorithms and evaluation. *Euro-Par 2005, LNCS 3648, Lisbon, Portugal*, 2005.

[MLM+05] P.J. Marron, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. Tinycubus: A flexible and adaptive framework for sensor networks. *EWSN 2005, Istanbul, Turkey*, 2005.

[MOD] Modoc - model-driven development of self-organizing control applications. http://kbs.cs.tu-berlin.de/projects/modoc.htm.

[MSK+05] C. Mallanda, A. Suri, V. Kunchakarra, S. S. Iyengar, R. Kannan, and A. Durresi. Simulating wireless sensor networks with omnet++. *submitted to IEEE Computer*, 2005.

[Nic06] Stefano Niccolai. Evaluation of two different schemes for subscriptions and notifications disseminated in a component-based publish/-subscribe system for wsns. Master's thesis, University of Pisa, June 2006.

[ns2] The ns manual (formerly ns notes and documentation).

[ns206] Network Simulator 2 Internet: www.nsnam.isi.edu/nsnam/index.php, 2006.

[PBM+04] J. Pollet, D. Blazakis, J. McGee, D. Rusk, and J. Baras. Atemu: A fine-grained sensor network simulator. *Proceedings of SECON'04, First IEEE Communications Society Conference on Sensors and Ad HOC Communications and Networks*, 2004.

[PLS02] S. Park, I. Locher, and M. Srivastava. Design of a wereable sensor badge for smart kindergarten. In *6th International Symposium on Wereable Computers (ISWC2002), Seattle, WA*, October 2002.

[PSC05] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks, Los Angeles, California*, page 48, Piscataway, NJ, USA, 2005. IEEE Press.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms*, November 2001.

[RFMB04] K. Römer, C. Frank, P. J. Maron, and C. Becker. Generic role assignment for wireless sensor networks. *11th ACM SIGOPS European Workshop. Leuven, Belgium*, 2004.

[SAJG00] K. Sohrabi, V. Ailawadhi, J.Gao, and G.Pottie. Protocols for self-organization of a wireless sensor network. *Personal Communication Magazine 7*, 2000.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys, Vol. 25, No. 1*, 1993.

[SK00] L. Subramanian and R.H. Katz. An architecture for building self-configurable systems. *MobiHoc'00. Boston,USA*, 2000.

[SM01] S. Slijepcvevic and M.Potkonjak. Power efficient organization of wireless sensor networks. *ICC'01, Helsinki, Finland*, 2001.

[SRWV05] J. Schiller, H. Ritter, R. Winter, and T. Voigt. Scatterweb - low power sensor nodes and energy aware routing. *System Sciences, 2005. HICSS '05. Proceedings of the 38th Annual Hawaii International Conference*, page 286c, 2005.

[TLP05] B.L. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. *Proceedings of IPSN'05,*

Fourth International Conference on Information Processing in Sensor Networks, 2005.

[TS07] Andrew S. Tanenmbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2007.

[WLLP01] B. Warneke, M. Last, B. Liebowitz, and KSJ Pister. Smart dust: communicating with a cubic-millimeter computer. *Computer*, 34:44–51, January 2001.

[WPJM06] Torben Weis, Helge Parzyjegla, Michael A. Jaeger, and Gero Mühl. Self-organizing and self-stabilizing role assignment in sensor/actuator networks. *The 8th International Symposium on Distributed Objects and Applications (DOA 2006)*, 4276:1807–1824, October 2006.

[ZBG98] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: A library for parallel simulation of large-scale wireless networks. *Workshop on Parallel and Distributed Simulation*, 1998.