Comparison of Distributed Optimization Algorithms in Sensor Networks

Simulations and results

CESARE MARIA CARRETTI

Masters’ Degree Project
Stockholm, Sweden January 2008

XR-EE-RT 2008:001
Abstract

We consider a peer-to-peer approach to wireless sensor networks using the IEEE 802.15.4 standard, with sensors not synchronized, and without any routing protocol. Only communications between neighbors are allowed. In this scenario we do a comparison between four distributed algorithms that solve a special class of optimization problems, which are of great interest in networking applications. We want to retrieve, without a central node, the average of a scalar measurement from all sensors in the network. In the final state, each sensor should have the global average of the considered measurement. To evaluate performances of the algorithms, we build an application for the network simulator ns2, and we do several simulations to evaluate convergence delay, and final error, respect to time and to number of packets sent. In this thesis we present the algorithms from theoretical and practical point of views, we describe our application for ns2, and we show the results obtained, which show that this types of algorithms, if tuned properly, work well and are ready to be practically implemented in a real sensor network.

Contents

1 Introduction ............................ 7
   1.1 General description .......... 8
   1.2 Distributed vs Centralized .... 9
   1.3 Originality of the work .... 9
   1.4 Thesis’ overview .......... 10
   1.5 Previous Work ........ 11

2 Background ................................ 13
   2.1 Wireless Sensor Networks ...... 14
      2.1.1 Hardware and Software ... 14
      2.1.2 Application of Wireless Sensor Networks ... 16
## Contents

2.2 IEEE 802.15.4 and ZigBee .................................. 18
  2.2.1 Physical layer (PHY) ................................... 19
  2.2.2 MAC sublayer ........................................... 20
  2.2.3 Devices for IEEE 802.15.4 ............................... 22
  2.2.4 Power saving ........................................... 22

3 Optimization Theory ........................................... 23
  3.1 Convex Optimization Problem ................................. 24
    3.1.1 Convex Optimization overview .......................... 24
    3.1.2 Dual approach ........................................ 25
    3.1.3 Solve the problem: descent methods ................... 27
  3.2 Distributed algorithms .................................... 29
    3.2.1 Distributed optimization problem ....................... 29
    3.2.2 Average: least squares solution ....................... 30
    3.2.3 Topology ............................................. 30
  3.3 Algorithms and naming ..................................... 31
    3.3.1 Unicast .............................................. 31
    3.3.2 Broadcast ............................................. 33
    3.3.3 Unreliable broadcast .................................. 36
    3.3.4 Dual .................................................. 37
  3.4 Robust Optimization ........................................ 39

4 Implementation .................................................. 41
  4.1 ns2 ....................................................... 42
    4.1.1 Structure and functioning of ns2 .................... 42
    4.1.2 802.15.4 in ns2 ...................................... 46
    4.1.3 New agent for ns2 .................................... 49
    4.1.4 OTcl script and simulation run ....................... 53
  4.2 Our Application: CCsensorApp ............................... 57
    4.2.1 CCsensorApp’s general structure ..................... 57
    4.2.2 Implementation of the algorithms ..................... 59
    4.2.3 Neighbors’ discovery ................................ 66

5 Results and Comments ........................................... 67
  5.1 Types of comparison ..................................... 68
    5.1.1 Tools ................................................ 68
  5.2 Algorithms ................................................ 70
    5.2.1 Parameter setting and tuning .......................... 70
    5.2.2 General comparison .................................... 71
    5.2.3 Broadcast Algorithm .................................. 75
    5.2.4 Broadcast Unreliable Algorithm ....................... 80
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.5 Dual Optimization Algorithm</td>
<td>84</td>
</tr>
<tr>
<td>5.2.6 Unicast Algorithm</td>
<td>86</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>91</td>
</tr>
<tr>
<td>7 Future work</td>
<td>95</td>
</tr>
<tr>
<td>Bibliography</td>
<td>97</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction
1.1 General description

During the last years, the diffusion of wireless sensor networks is making the research concentrate a lot on this field. Also, the continuous improvements in sensors’ technology give unlimited possibility of application: environment monitoring, home automation, traffic control, assistance, without mentioning military application, are all very actual and useful fields of application for sensor networks, and there is in each of these fields a big effort in researching ways to improve the performances of the networks, to find better ways to do the same things, to discover new applications. The use of sensors could, for example, go from avoid the use of wires in simple applications like remote control (e.g., a light switcher), to built a large complex sensor network to monitor the pollution in the atmosphere.

The request of a wireless sensor network are very different from those of a computer wireless network, and from those of mobile phones: sensors do not need high data rate transfer, while need data reliability, power saving and network self-configuring. In particular the problem of preserving power is very important, because of the difficulty and, often, impossibility, in changing the battery pack. For all these reasons, there was the need to develop a new protocol especially thought for low data rate networks. The answer was the IEEE 802.15.4 standard. The aim of this standard is indeed to provide a communication protocol for networks that not need an high bit rate and that have power consumption constraints.

When a new technology is introduced, to exploit it at best, there is the need of doing a lot of research upon the different aspects and the different possibilities this technology can offer. This is what is happening for IEEE 802.15.4 and for sensor networks in general. The scientific world is showing a big interest on these issues, thinking that in the next future, there will be an exponential growth of wireless sensor networks in a lot of different fields.

This Master Thesis Project aims at giving a contribute to the research, simulating on the network simulator ns2 a wireless sensor network (which use the standard IEEE 802.15.4) and implementing over it four distributed optimization algorithms, to calculate the average of sensors’ scalar measurement using only peer-to-peer connections, without any routing protocol and without synchronization. This approach is very useful to have high reliability in the system, and to preserve as much as possible battery power inside devices. A project for future work is to implement the algorithms in a real sensor network, using t-mote sky sensors, but it is important to test in a simulation what kind of performances to expect and the possible problems.
1.2 Distributed vs Centralized

The first thing people who want to calculate the average of scalar measurements in a sensor network think, is to send the values from each sensor to a central node who does the sum of all values and then divides for the total number of sensors in the network. This approach has two main disadvantages:

- Reliability, because if there is a failure in the central node, then all network would be out of order
- With the assumption that the central node is not in range of all other nodes, there is the problem of multi-hop passages, with the nodes closer to the central one that have to transmit their data and data from far nodes, in this way consuming more energy

For these reasons, the approach in this work is to utilize only peer-to-peer communications, without any multi-hop passages, and without any central node. In this way there are the advantages of power saving, of better reliability of the network, because in case a node has a failure the network is able to reconfigure itself, and also that each node, after some iterations of the algorithms, should have the global value of convergence, so it would be possible to retrieve data regarding all the network from anyone of the nodes. It is possible to think to applications as environmental control, where the sensors are spread, e.g., in a forest for fire control, or for general parameter control. Another interesting field for the kind of networks we take into consideration in this work could be road monitoring, e.g., in highways, cars could have at the same time local and global informations, connecting with sensors lying on the road side. These sensors could be divided into interconnected clusters, with optimization algorithms inside each cluster and between all clusters. We talk more about possible applications in section 2.1.2.

1.3 Originality of the work

There are a lot of works in literature about wireless sensor networks (see section 1.5), but usually the approach is only theoretical or with Matlab simulations. This work shows a comparison between different distributed optimization algorithms using the network simulator ns2, a more realistic environment than Matlab. The simulator ns2, as deeply explained in section 4.1, takes in account a lot of physical variables like power in transmission, height of antenna from the ground, propagation model, retransmissions, propagation upon the channel. Of course the best way to try these algorithms would be to implement them on real sensors (see future work section at the end of this thesis),
but before trying to spend a lot of time and resources on real networks, there is the need of doing some simulations, to see whether the research is worthy to be done. We think it is in our case.

1.4 Thesis’ overview

After this introduction and a description about previous works in the same field of this work (in the next section), in the second chapter we provide a background for wireless sensor networks (WSN): what is a WSN, what types of hardware exist in the industry, what are the main fields of application, and also a description of the standard communication protocol used in WSN, the IEEE 802.15.4. In the third chapter, we present the theory behind the four algorithms: it is given a general introduction about convex optimization, and then a description of distributed optimization, with details about each algorithm from a mathematical point of view. In the forth chapter, we describe the implementation: what is and how to use in general the network simulator ns2, how to build an application for it, how to change the source code, how our application work and problems in the implementation of the algorithms. In the fifth chapter, we present our results, with tables, graphs and comments. We describe the metrics and the tools used to evaluate the performances. First is showed a general comparison, and then a particular list of results for each algorithms, varying also different parameters. Finally, in the chapters sixth and seventh respectively, we give some conclusions and description about something we would like to do in the future to expand this work.
1.5  Previous Work

Our work touches different fields, all quite present in the literature. Regarding optimization in general, the literature is huge. For us was very useful the books of S. Boyd [8], and the book from M. Fischetti [13] (in Italian). In particular about subgradients methods, are very important the works of A. Nedić with D. P. Bertsekas, [22, 21], used as basis in many other works (e.g., [17, 23]).

Regarding distributed optimization, particularly important for our work, there are the works of M. Rabbat with R. Nowak, [27, 26, 25], where, in particular, [25] forms the basis for distributed optimization in sensor networks. They investigate a general class of distributed algorithms for ”in-network” data processing, eliminating the need to transmit raw data to a central point. This is the basis for several followings works, such as the ones cited below (in particular [17, 31, 37, 18]).

In [17], B. Johansson, M. Rabi and M. Johansson propose a distributed algorithm for sensor networks (it is one of the algorithms implemented in our work, we describe it in details in section 3.3.1). As basis for our implementation, we used also the paper from Xiao and Boyd [36], to implement, with some differences (see for details section 3.3.2), their algorithm, and the paper from Rabbat, Nowak, and Bucklew, [26], whose algorithm is implemented almost exactly in our simulations (see section 3.3.4 for details). We can find an approach similar to [25] in the work of S. H. Son et others, [31], but here they have an in cluster approach, to be able to have a trade off between accuracy of the estimation and power constraints, using the subgradient method within each cluster. Each cluster, then, can send the convergence value to a central station. It is possible to choose how many clusters to form in the network (more clusters mean less power consume but also less estimation accuracy). In [9], Intanagonwiwat, Govindan, and Estrin study a data-centric paradigm, called ”Directed diffusion”. It is something different from what we do, but it is very interesting: all nodes in a directed diffusion-based network are application aware. They apply this model to a simple remote-surveillance sensor network.

Another important field of application for distributed algorithm in sensor networks is robust estimation: it is beyond the scope of our work but it is very correlated, and there are several paper about it, e.g., the already cited [31] and [37]: Xiao, Boyd, and Lall propose here a simple distributed iterative scheme, based on distributed average consensus in the network, to compute the maximum-likelihood estimate of the parameters, diffusing informations across the network by updating each node’s data with a weighted average of its neighbors’ data. At each step, every node can compute a local weighted least-squares estimate, which converges to the global maximum-likelihood so-
olution. This scheme is robust to unreliable communication links. Also about robustness, Kar and Moura, in [18], study the impact of the topology of a sensor network on distributed average consensus algorithms when the network links fail at random, deriving convergence results.

Even if it is not directly aimed at sensor networks, there is an interesting approach in [19]: they introduce a reliable neighborcast protocol (RNP), for transferring data reliably in networks with nodes that change their positions and availability in a dynamic way. That protocol is at application layer in the OSI model, so it is possible to use it virtually over any different physic and MAC layer (so also with IEEE 802.15.4). The problems with RNP applied to sensors could concern power constraints of sensors (in fact RNP needs several communications between nodes) and sensors’ CPU too small computational capability.

All these work, more or less, describe the problem from a theoretical point of view. What we want to do in this thesis is to try to do a more realistic simulation of distributed optimization in sensor networks, using the ns2 network simulator.
Chapter 2

Background
CHAPTER 2. BACKGROUND

2.1 Wireless Sensor Networks

A Wireless Sensor Network (WSN) is a network consisting of a large number of heterogeneous sensor devices spread over a large field. Sensors can monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations. Each node in a sensor network, is usually equipped with a radio transceiver or other wireless communications device, a small microcontroller, and an energy source, usually a battery. Size of sensor node can vary from the size of a small box to the size of a grain of dust. In this latter case, it is used the name smart dust: sensors so small to result almost invisible, but with interconnection and measuring capabilities. They are mainly useful in military applications but also in other control applications. In general, it is possible to find some common, unique characteristics that a WSN should have:

- Small-scale sensor nodes
- Limited power they can harvest or store
- Harsh environmental conditions
- Node failures
- Mobility of nodes
- Dynamic network topology
- Communication failures
- Heterogeneity of nodes
- Large scale of deployment
- Unattended operation

2.1.1 Hardware and Software

In general, sensor nodes can be imagined as small computers, extremely basic in terms of their interfaces and their component. There are a cpu, a memory slot, a communications chipset, and a power source, usually a battery. In figure 2.1 it is shown an example of sensor device: the tmote sky from moteiv (now sentilla, [30]). For deep details, refer to the tmote sky data-sheet [23], but here we provide some key features of this very diffuse type of sensor, from its data-sheet, also to give a general idea on what type of hardware capabilities a sensor needs:
2.1. WIRELESS SENSOR NETWORKS

Figure 2.1: An example of sensor device: the tmote sky.

- 250kbps 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver
- Interoperability with other IEEE 802.15.4 devices
- 8MHz Texas Instruments MSP430 microcontroller (10k RAM, 48k Flash)
- Integrated ADC, DAC, Supply Voltage Supervisor, and DMA (Direct Memory Access) Controller
- Integrated onboard antenna with 50m range indoors / 125m range outdoors
- Integrated Humidity, Temperature, and Light sensors
- Ultra low current consumption
- Fast wakeup from sleep (< 6\(\mu\)s)
- Hardware link-layer encryption and authentication
- Programming and data collection via USB
- 16-pin expansion support and optional SMA (SubMiniature version A) antenna connector
- TinyOS support: mesh networking and communication implementation
- Complies with FCC (Federal Communication Commission) Part 15 and Industry Canada regulations

Every sensor, in general, has a transceiver, a microcontroller, memory, and a power source. We describe the communication standard IEEE 802.15.4 in section 2.2.
The problem in writing software for sensors is in severe limitations with respect to memory, power consumption, and computations capabilities. The most diffused operating system for sensors is TinyOs, an open-source operating system designed for wireless embedded sensor networks. Other possibilities are in the Contiki operative system. This is an OS developed not specifically for sensors, but constrained systems or ‘deeply embedded’ systems in general. These two (TinyOs and Contiki), are the most diffused Operative System used on sensors such as the tmote sky in figure 2.1 but there are also a lot of other solutions in property software for embedded system. The market seems to go towards an easier way to develop applications for sensors, e.g., using a programming language as java instead of C or similar (e.g., nesC on TinyOs).

2.1.2 Application of Wireless Sensor Networks

Figure 2.2: Examples of possible applications for a Wireless Sensor Networks.

In figure 2.2 we provide some examples of typical possible applications for a WSN such as the one we consider in this thesis. In the picture, we want to stress the fact that a WSN should interact with a lot of different other networks and devices, such as smart phones and gateways to control remotely the sensors and to retrieve data from them. Applications are not limited:
from military battlefield surveillance, to environmental monitoring (e.g., in
a forest or in a vulcan, or in the atmosphere), from traffic control and road
monitoring, to smart factories, to improve and monitor in a way not possible
until now production processes. Other very useful applications could regard
people assistance and home automation. With the decrease of the devices’
size and increase of battery duration, there will be an exponential growth of
fields which could take advantages by the use of WSNs.
2.2 IEEE 802.15.4 and ZigBee

A sensor network is a type of Low Rate Wireless Personal Area Network (LR-WPAN), a simple, low-cost communication network that allows wireless connectivity in applications with limited power and low throughput requirements. IEEE 802.15.4 is the standard which specifies the physical layer and Medium Access Control, (MAC) for devices belonging to a LR-WPAN, while ZigBee is the name of a specification for a suite of high level communication protocols (mainly network and application layers) in the WPAN (see figure 2.3).

Our work was to simulate, using the network simulator ns2 (see section 4.1.1), four distributed optimization algorithms over the IEEE 802.15.4 protocol. We did not use ZigBee, because for our work we did not need a routing protocol like AODV (Ad-hoc On-demand Distance Vector), one of the main component of ZigBee standard. We was interested in only peer-to-peer communications, without any routing protocol, synchronization, and beacons, to test the behavior of the algorithms in this situation.

The standard IEEE 802.15.4 defines two topologies: the star topology and the peer-to-peer topology. In the star topology the communication is established between devices and a single central controller, called PAN coordinator. In the peer-to-peer topology there is also a PAN coordinator, but, differently from the star topology, any device may communicate with any other device as long as they are in range of one another. While in the star topology the PAN coordinator is already decided, in the peer-to-peer case one device is nominated as the PAN coordinator, for instance, by virtue of being the first device to communicate on the channel. In order to have communication in the peer to peer mode, the devices wishing to communicate will need to either receive constantly (and transmit its data using unslotted CSMA/CA, described in section 2.2.2) or synchronize with each other (see also [15, section 5.3.2 and 5.5.2.3]). For the kind of work we did, we needed to use only the peer-to-peer topology without synchronization, which allows more complex network formations to be implemented respect to the star topology.

As we said above in this section, IEEE 802.15.4 specifies only the physical layer and the MAC sublayer (following the layout of the OSI model). Each layer is responsible for one part of the standard and offers services to the higher layers. An IEEE 802.2 Type 1 logical link control (LLC), the standard higher sublayer of the data link layer, can access the MAC sublayer through the service-specific convergence sublayer (SSCS, see figure 2.3), a direct interface to the primitive of the MAC. Here we describe, respectively in section 2.2.1 and 2.2.2, the two components of the standard, Physical layer and Medium Access Control sublayer.
2.2.1 Physical layer (PHY)

The PHY, provides two services (see the IEEE 802.15.4 standard definition [15 section 5.4.1 and 6] and also the works of J. Zheng with M. Lee, [40, 39]): the PHY data service and the PHY management service interfacing to the physical layer management entity (PLME) service access point (SAP). The PHY data service enables the transmission and reception of PHY protocol data units (PPDUs) across the physical radio channel.

The feature of the PHY are (for details, see [15 chapter 6]):

- activation and deactivation of the radio transceiver
• energy detection within the current channel (part of channel selection algorithm)
• link quality indicator (LQI) for received packets (a characterization of the strength and/or quality of a received packet)
• channel frequency selection
• clear channel assessment (CCA) for carrier sense multiple access with collision avoidance (CSMA/CA, see section 2.2.2)
• transmitting as well as receiving packets across the physical medium

The radio operates at one or more of the following unlicensed ISM (Industrial Scientific Medical) bands:

• 868-868.6 MHz (e.g. Europe) - 1 channel with a data rate of 20 kb/s (3 channels in the newest version of the standard)
• 902-928 MHz (e.g. North America) - 10 channels with a data rate of 40 kb/s (30 channels in the newest version of the standard)
• 2400-2483.5 MHz (worldwide) - 16 channels with a data rate of 250 kb/s for a total of 27 channels (59 channels in the newest version of the standard).

An IEEE 802.15.4 network can choose to work in one of the channels depending on the availability, congestion state, and data rate of each channel. Different data rates offer better choices for different applications in terms of energy and cost efficiency, because if there is no need to have the highest data rate, the network can choose to work in a lower channel, saving power (because of the lower rate and for the lower frequency).

2.2.2 MAC sublayer

The MAC sublayer provides two services (see the IEEE 802.15.4 standard definition [15, section 5.4.2 and 7] and also, again, the works of J. Zheng with M. Lee, [40, 39]): the MAC data service and the MAC management service interfacing to the MAC sublayer management entity (MLME) service access point (SAP). The MAC data service enables the transmission and reception of MAC protocol data units (MPDUs) across the PHY data service.

The features of the MAC sublayer are (for details, see [15, chapter 7]):

• beacon\(^1\) management, if the device is a coordinator

---

\(^1\)Is called *beacon* a periodical signal, provided by the network coordinator, which gives synchronization to all devices in the network.
2.2. IEEE 802.15.4 AND ZIGBEE

- synchronizing to network beacons
- channel access through CSMA/CA mechanism or Guaranteed Time Slot (GTS)\(^2\)
- handling and maintaining the GTS mechanism
- frame validation
- acknowledged frame delivery (see section 2.2.2)
- supporting PAN association and disassociation
- providing a reliable link between two peer MAC entities

In our work we considered the situation of a non beacon-enabled network. For channel access, like most other protocols designed for wireless networks, IEEE 802.15.4 uses CSMA/CA mechanism, slotted or unslotted, depending whether beacon enabled mode or non-beacon enabled mode (as in our case) is used. However, the standard does not include the request-to-send (RTS) and clear-to-send (CTS) mechanism as in the IEEE 802.11 standard, in consideration of the low data rate used in LR-WPANS.

CSMA/CA mechanism

CSMA/CA, *Carrier Sense Multiple Access With Collision Avoidance*, is a multiple access protocol. In normal CSMA, a node who has data to transmit, listen to the channel for a certain amount of time, to check activity on the channel, through Clear Channel Assessment (CCA) mechanism. If the channel is sensed to be *idle*, then the node transmits. If the channel is *busy*, the node has to defer its transmission. In the CSMA/CA, after the channel is sensed *idle*, the node waits for a random period. If the channel is found to be idle, following the random backoff, the node transmits its data. If the channel is *busy*, following this random backoff, the node waits for another random period before trying to access the channel again. In IEEE 802.15.4 acknowledgment frames, used to have reliable communication in unicast transmissions, are sent without using a CSMA/CA mechanism (see [15] section 5.5.4.1). It is possible to disable the acknowledgment frames, if there is already some mechanism who grants reliability at higher layers, or if reliability is not requested.

\(^2\)A guaranteed time slot is a feature for beacon enabled networks, to better guarantees real time communications by giving to a device some slots to send data avoiding the CSMA/CA mechanism.
2.2.3 Devices for IEEE 802.15.4

Two different types of devices are defined in a IEEE 802.15.4 network, a full function device (FFD) and a reduced function device (RFD). A FFD can talk to RFDs and other FFDs, and operate in three modes serving either as a PAN coordinator, a coordinator, or a device. An RFD can only talk to an FFD and is intended for applications that are extremely simple. They do not have the need to send large amounts of data and may only associate with a single FFD at time. The association process is the service used to establish membership for a device in a WPAN, in this case to let the FFD know about the presence of the RFD. In our work we considered all nodes as FFDs.

2.2.4 Power saving

One of the most important issues for WSN is the power saving: as we said in the introduction, it is very difficult, often impossible, to change the battery packs to sensors spread in an environment. To save power, each sensor is able to put itself in a sleep mode, and therefore have a duty-cycle, listening to the channel not always, but at precise time, or only when requested. In this thesis, to be more general, we take in consideration the worst case, with sensor listening all the time and without any beacon and synchronization. The IEEE 802.15.4 standard (see [15]) provides for duty-cycles and beacons. There are several studies on how to improve MAC to save power. This issues is beyond the scope of this thesis, but we remind to [14] for details: they compare CSMA/CA mechanism with others to find a better way to respect power constraints.
Chapter 3

Optimization Theory
3.1 Convex Optimization Problem

Our work was for the most part a practical one, but it is important to understand the theoretical basis for the algorithms we implemented. The mathematical basis is convex optimization theory, in fact we will approach the problem from this point of view. Indeed we want to solve the problem

$$ \min_{\theta} \sum_{i=1}^{N} f_i(\theta) \quad \text{subject to} \quad \theta \in \Theta, $$

(3.1)

where $\theta$ is the variable to optimize, $\Theta$ is the feasible set for $\theta$, and $f_i(\theta)$ is the cost function associated with node $i$, function assumed to be a convex one, so that the local minimum is the global minimum (see [8] for references about convex optimization).

3.1.1 Convex Optimization overview

We here give a general overview about convex optimization and solution algorithms. For this introduction, we will follow almost exactly chapters four, five, and nine of Boyd’s Convex Optimization book [8]. For the subgradient part we follow [21] and [11].

In general, an optimization problem, not necessarily convex, can be written as

$$ \min_{x} \ f_0(x) \quad \text{subject to} \quad f_i(x) \leq 0, \ i = 1, \ldots, m $$
$$ h_i(x) = 0, \ i = 1, \ldots, p $$

(3.2)

where the problem is finding an $x \in \mathbb{R}^n$ (optimization variable) that minimizes $f_0(x)$ among all $x$ that satisfy the conditions $f_i(x) \leq 0, i = 1, \ldots, m$ and $h_i(x) = 0, i = 1, \ldots, p$.

$f_0 : \mathbb{R}^n \to \mathbb{R}$ is the objective function or cost function. The inequalities $f_i(x) \leq 0$ are called the inequality constraints, and the corresponding functions $f_i : \mathbb{R}^n \to \mathbb{R}$ are called the inequality constraint functions. The equations $h_i(x) = 0$ are called the equality constraints, and the functions $h_i : \mathbb{R}^n \to \mathbb{R}$ are the equality constraint functions. The domain of the optimization problem (3.2) is the set of points for which the objective and all constraint functions are defined,

$$ \mathcal{D} = \bigcap_{i=0}^{m} \text{dom } f_i \cap \bigcap_{i=1}^{p} \text{dom } h_i $$

A point $x \in \mathcal{D}$ is feasible if it satisfies the constraints $f_i(x) \leq 0, i = 1, \ldots, m$, and $h_i(x) = 0, i = 1, \ldots, p$. The problem (3.2) is said to be feasible
3.1. CONVEX OPTIMIZATION PROBLEM

if there exists at least one feasible point, and infeasible otherwise. The set of all feasible points is called the feasible set or the constraint set. The optimal value \( p^* \) of the problem (3.2) is defined as

\[
p^* = \inf \{ f_0(x) | f_i(x) \leq 0, i = 1, \ldots, m, h_i(x) = 0, i = 1, \ldots, p \}.
\]

A convex optimization problem is one of the form (as we can find in section 4.2 of [8])

\[
\begin{align*}
\text{minimize} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 0, \quad i = 1, \ldots, m \\
& \quad a_i^T(x) = b_i, \quad i = 1, \ldots, p,
\end{align*}
\]

where \( f_0, \ldots, f_m \) are convex functions\(^1\).

The equation (3.3) differs from the (3.2) on three additional requirements:

- the objective function must be convex,
- the inequality constraint functions must be convex,
- the equality constraint functions \( h_i(x) = a_i^T x - b_i \) must be affine\(^2\).

The feasible set of a convex optimization problem is convex, since it is the intersection of the domain of the problem

\[
D = \bigcap_{i=0}^{m} \text{dom} f_i,
\]

which is a convex set, with \( m(\text{convex}) \) sublevel sets \( \{x | f_i(x) \leq 0\} \) and \( p \) hyperplanes \( \{x | a_i^T x = b_i\} \). Thus, in a convex optimization problem, we minimize a convex objective function over a convex set.

### 3.1.2 Dual approach

If we have an optimization problem like in the equation (3.2), we could think about relaxing some constraints, considering them ‘inside’ the objective function (see also [11] (in Italian) and [8, chap. 5]). In practice we take the constraints into account by augmenting the objective function with a weighted

---

\(^1\)A function is convex if satisfies \( f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y) \) for all \( x, y \in \mathbb{R}^n \) and all \( \alpha, \beta \in \mathbb{R} \) with \( \alpha + \beta = 1, \alpha \geq 0, \beta \geq 0 \).

\(^2\)A function \( f : \mathbb{R}^n \to \mathbb{R}^m \) is affine if it is a sum of a linear function and a constant, i.e., if it has the form \( f(x) = Ax + b \), where \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \).
sum of the constraint functions. As in \cite[sections 5.1.1-5.2.2]{S}, we define the \textit{Lagrangian} \( L : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R} \) associated with the problem (3.2) as
\[
L(x, \lambda, \nu) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x),
\]
with \( \text{dom } L = D \times \mathbb{R}^m \times \mathbb{R}^p \). We refer to \( \lambda_i \) as the \textit{Lagrange multiplier} associated with the \( i \)th inequality constraint \( f_i(x) \leq 0 \); similarly we refer to \( \nu_i \) as the Lagrange multiplier associated with the \( i \)th equality constraint \( h_i(x) = 0 \). The vectors \( \lambda \) and \( \nu \) are called the \textit{dual variables} or \textit{Lagrange multiplier vectors} associated with the problem (3.2).

We define the \textit{Lagrange dual function} \( g : \mathbb{R}^m \times \mathbb{R}^p \to \mathbb{R} \) as the minimum value of the Lagrangian over \( x \) for \( \lambda \in \mathbb{R}^m, \nu \in \mathbb{R}^p \):
\[
g(\lambda, \nu) = \inf_{x \in D} \left( f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x) + \sum_{i=1}^{p} \nu_i h_i(x) \right).
\]
When the Lagrangian is unbounded below in \( x \), the dual function takes on the value \(-\infty \). Since the dual function is the pointwise infimum of a family of affine functions of \( (\lambda, \nu) \), it is concave\(^3\), even when the problem (3.2) is not convex.

It can easily verified (see \cite[section 5.1.3]{S} and \cite{L}) that the Lagrange dual function yields lower bounds on the optimal value \( p^* \) of the problem (3.2): For any \( \lambda \succeq 0 \) and any \( \nu \) we have
\[
g(\lambda, \nu) \leq p^*.\tag{3.5}
\]

The best lower bound that can be obtained from the Lagrange dual function is the \textit{Lagrange dual problem} associated with the problem (3.2):
\[
\begin{align*}
\text{maximize} & \quad g(\lambda, \nu) \\
\text{subject to} & \quad \lambda \succeq 0.
\end{align*}
\]

The original problem (3.2) is sometimes called the \textit{primal problem}. We refer to \( (\lambda^*, \nu^*) \) as \textit{dual optimal} or \textit{optimal Lagrange multipliers} if they are optimal for the problem (3.6). The Lagrange dual problem (3.6) is a convex optimization problem, since the objective to be maximized is concave and the constraint is convex. This is the case whether or not the primal problem (3.2) is convex.

\(^3\)A function \( f \) is \textit{concave} if \(-f\) is convex.
3.1.3 Solve the problem: descent methods

A problem like (3.3), usually has to be solved by an iterative algorithm, i.e. an algorithm that computes a sequence of points \(x^{(0)}, x^{(1)}, \ldots \in \text{dom } f\) with \(f(x^k) \to p^*\) as \(k \to \infty\). Such a sequence of points is called a minimizing sequence for our problem. The algorithm is terminated when \(f(x^{(k)}) - p^* \leq \epsilon\), where \(\epsilon > 0\) is some specified tolerance (see [8, section 9.1 and 9.2]). In general the upgrade step is similar to

\[x^{(k+1)} = x^{(k)} + t^{(k)} \Delta x^{(k)}\]

with \(t^{(k)} > 0\) (except when \(x^{(k)}\) is optimal). The concatenated symbols \(\Delta\) and \(x\) that form \(\Delta x\) mean a vector in \(\mathbb{R}^n\) called the step or search direction (even though it needs not have unit norm), and \(k = 0, 1, \ldots\) denotes the iteration number. The scalar \(t^{(k)} \geq 0\) is called the step size or step length at iteration \(k\) (even though it is not equal to \(\|x^{(k+1)} - x^{(k)}\|\) unless \(\|\Delta x^{(k)}\| = 1\)). In a descent method,

\[f(x^{(k+1)}) < f(x^{(k)})\]

except when \(x^{(k)}\) is optimal. This implies that for all \(k\) we have \(x^{(k)} \in S\), the initial sublevel set\(^4\), and in particular we have \(x^{(k)} \in \text{dom } f\). From convexity we know that \(\nabla f(x^{(k)})^T (y - x^{(k)}) \geq 0\) implies \(f(y) \geq f(x^{(k)})\), so the search direction in a descent method must satisfy

\[\nabla f(x^{(k)})^T \Delta x^{(k)} < 0,\]

i.e., it must make an acute angle with the negative gradient. We call such a direction a descent direction (for \(f\), at \(x^{(k)}\)).

The outline of a general descent method is as follows. It alternates between two steps: determining a descent direction \(\Delta x\), and the selection of a step size \(t^{(k)}\).

---

\(^4\)The \(\alpha\)-sublevel set of a function \(f : \mathbb{R}^n \to \mathbb{R}\) is defined as

\[C_\alpha = \{x \in \text{dom } f | f(x) \leq \alpha\}.
\]

Sublevel sets of a convex function are convex, for any value of \(\alpha\). The proof is immediate from the definition of convexity: if \(x, y \in C_\alpha\), then \(f(x) \leq \alpha\) and \(f(y) \leq \alpha\), and so \(f(\theta x + (1 - \theta)y) \leq \alpha\) for \(0 \leq \theta \leq 1\), and hence \(\theta x + (1 - \theta)y \in C_\alpha\) (see [8 section 3.1.6]).
Algorithm 1 General descent method.

Require: a starting point \( x \in \text{dom } f \).

1: repeat
2: Determine a descent direction \( \Delta x \)
3: Line search. Choose a step size \( t > 0 \).
4: Update. \( x := x + t \Delta x \).
5: until stopping criterion is satisfied.

The third step is called the line search since selection of the step size \( t \) determines where along the line \( \{ x + t \Delta x | t \in \mathbb{R}^+ \} \) the next iterate will be.

Gradient descent method

We first consider the problem minimize \( f(x) \). This is the case of an unconstrained optimization problem, where \( f : \mathbb{R}^n \to \mathbb{R} \) is convex and twice continuously differentiable, which implies that \( \text{dom } f \) is open (see [8, section 9.1]).

Since \( f \) is differentiable and convex, a necessary and sufficient condition for a point \( x^* \) to be optimal (if the problem is solvable) is

\[
\nabla f(x^*) = 0
\]

We denote the optimal value, \( \inf_x f(x) = f(x^*) \), as \( p^* \).

Knowing that, a natural choice for the search direction in algorithms like the General descent method is the negative gradient \( \Delta x = -\nabla f(x) \). We can rewrite the algorithm in this way:

Algorithm 2 Gradient descent method.

Require: a starting point \( x \in \text{dom } f \).

1: repeat
2: \( \Delta x := -\nabla f(x) \).
3: Line search. Choose a step size \( t > 0 \).
4: Update. \( x := x + t \Delta x \).
5: until stopping criterion is satisfied.

Subgradient

If \( f(x) \) is convex but not differentiable, we cannot calculate the \( \nabla f(x) \) and therefore we do not have a suitable search direction. Here it is useful to introduce the concept of subgradient.
First we recall an important property of the gradient of a convex differentiable function (see [25, section 2], [22], and [11]). For a convex differentiable function, $f : \mathbb{R}^n \to \mathbb{R}$, the following inequality for the gradient of $f$ at a point $x_0$ holds for all $x \in \text{dom } f$:

$$f(x) \geq f(x_0) + (x - x_0)^T \nabla f(x_0).$$

In general, for a convex function $f$, a subgradient of $f$ at $x_0$ (observing that $f$ may not be differentiable at $x_0$) is any direction $g$ such that

$$f(x) \geq f(x_0) + (x - x_0)^T g,$$

and the subdifferential of $f$ at $x_0$, denoted $\partial f(x_0)$, is the set of all subgradients of $f$ at $x_0$. Note that if $f$ is differentiable at $x_0$ then $\partial f(x_0) \equiv \{\nabla f(x_0)\}$, i.e., the gradient of $f$ at $x_0$ is the only direction satisfying (3.8).

### 3.2 Distributed algorithms

Our problem is to find a way to have reliable communications to make optimization algorithms converge in networks composed of several sensors spread over an environment (for example a forest or a building), while preserving as much power as possible. Preserving power is difficult using a centralized approach, because of multi-hop passages, so it is important to find a way for reaching convergence (hopefully with a low latency and quickly) in distributed sensor networks, that better can answer to power constraints. If we use a distributed algorithms, we also are more covered in case of node failures (in particular in the case of failure of the central node) and we can retrieve the values from every node of the network.

#### 3.2.1 Distributed optimization problem

Many estimation criteria possess the following important form (see [23]):

$$f(\theta) = \frac{1}{n} \sum_{i=1}^{n} f_i(\theta),$$

where $\theta$ is the parameter of function to be estimated, and $f(\theta)$ is the cost function which can be expressed as a sum of $n$ “local” functions $\{f_i(\theta)\}_{i=1}^{n}$ in which $f_i(\theta)$ only depends on the data measured at sensor $i$. So we can specialize our problem in the form (3.1) that we rewrite here for convenience:

$$\min_{\theta} \sum_{i=1}^{N} f_i(\theta)$$

subject to $\theta \in \Theta,$

(3.9)
where \( f_i(\theta) \) is the node \( i \) associated cost function, that we assume to be a convex function.

So we can spread the computation of the algorithms over a network, finally having in each node the same value, hopefully the optimum. The purpose of this thesis is to check through the network simulator ns2 (see section 4.1) several algorithms to diffuse data over a sensor network and to have a comparison between them. We would like to discover advantages and disadvantages of different solutions presented in the literature, more close to reality than only Matlab simulations. Of course the best would be to simulate with real sensors, but this approach is more expensive and less immediate, so it is better in our opinion to begin to work with real sensor only after several simulations that indicate the best ways in which focus the research, in a way as close as possible to reality.

### 3.2.2 Average: least squares solution

We concentrate our work on calculating the average because this is a problem of particular interest in literature (see for example [25, 18, 26, 20, 37]): if we have a network with several sensors spread in an environment, it is very natural to want to know the average of the values of each sensor, e.g., if we have temperature sensors spread in an environment, it would be very useful to retrieve, first of all, the average temperature in that environment.

From a convex optimization point of view, if we have a network of \( N \) sensors, the average is the minimization of the sum of squares \( \frac{1}{n} \sum_{i=1}^{N} (x_i - \theta)^2 \), where \( x_i \) is the local value of each sensor, and \( \theta \) is the real average that we want to estimate. It is a least squares solution. The local function that every sensor has to calculate is \( f_i(\theta) = (x_i - \theta)^2 \) and our problem is

\[
\begin{align*}
\text{minimize} & \quad \sum_{i}^{N} (x_i - \theta)^2 \\
\text{subject to} & \quad \theta \in \Theta,
\end{align*}
\]

We assume that \( \theta \) is scalar for simplicity, but the algorithms are readily extended to the vector case.

### 3.2.3 Topology

For the problem (3.9), we need to have a communication topology represented (see [17]) by a graph \( G = (V, E) \) with vertex set \( V = \{1, \ldots, N\} \) and edge set \( E \subseteq V \times V \). The presence of an edge \((i, j)\) in \( E \) means that node \( i \) can communicate directly with node \( j \), and vice versa. The graph \( G \) has \( N \) nodes (vertices). For all the algorithms, we made the assumption that the communication topology is connected.
3.3 Algorithms and naming

The most part of our work was to implement in practice through the network simulator ns2 (section 4.1) several distributed optimization algorithms to compute the average value of sensor networks with different topologies and different number of nodes. We did simulations and comparison for:

- Unicast subgradient distributed algorithm (called unicast in this thesis)
- Broadcast algorithm with acknowledgments (called broadcast in this thesis)
- Broadcast algorithm without acknowledgments (called unreliable broadcast in this thesis)
- Broadcast dual optimization algorithm without acknowledgments (called dual in this thesis)

To refer to the algorithms, our choice in this thesis is to simplify in the name of each algorithms describing the main feature of each algorithm in the name, in a way as easy as possible to understand and to remember, in particular in the section 5.2 in which we provide results.

Here we provide a mathematical background and general description for all this algorithms. For the implementation details of each one, see section 4.2.2 of this work. All algorithms converge to a value more or less close to the optimum one, depending from the algorithm and from parameters (see section 5.2).

3.3.1 Unicast

We implemented this algorithm in ns2 following almost literally the work of B. Johansson, M. Rabi, and M. Johansson in [17]. This algorithm extends the randomized incremental subgradient method with fixed stepsize due to Nedić and Bertsekas [22, 21]. Nodes maintain individual estimates and need to exchange information only with their neighbors. In a randomized version of the estimate passing scheme, a node sends the value to a random neighbor over the whole network, in a way that needs a routing protocol and that can produce multi-hop passages. In [17] they show that it is sufficient to send to a random neighbor, avoiding multi-hop passages. The transition probabilities, which determine how the parameter estimate is passed around in the network, can be computed using local network topology characteristics.
In [17] the optimization problem is defined as

$$\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{N} f_i(x_i, \theta) \\
\text{subject to} & \quad x_i \in \chi_i, \quad i = 1, \ldots, N \\
& \quad \theta \in \Theta,
\end{align*}$$

(3.10)

where $f_i(x_i, \theta)$ is a cost function associated with node $i$, $x_i$ is a vector of variables local to node $i$, and $\chi_i$ is the feasible set for the local variables. The set $\Theta$ is the feasible set of a global (network-wide) decision variable $\theta$. They assume also that $f_i$ are convex functions and that $\chi_i$ and $\Theta$ are convex sets with non-empty interior.

The problem (3.10) is then rewritten as

$$\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{N} q_i(\theta) \\
\text{subject to} & \quad \theta \in \Theta,
\end{align*}$$

(3.11)

where $q_i(\theta) = \min_{x_i \in \chi_i} f_i(x_i, \theta)$, defining then $q^*$ as the optimal value of (3.11) and $q(\theta) = \sum_{i=1}^{N} q_i(\theta)$. In [17], then, they assume that each (convex) component $q_i(\cdot)$ has a subgradient $g_i(\theta)$ at $\theta$ and that each of these subgradients are bounded as follows

$$g_i(\theta) \leq C \quad \text{for all } \theta \in \Theta \text{ and all } i = 1, \ldots, N.$$ 

The update equation of the estimate, $\theta^k$, of the optimizer is

$$\theta^{k+1} = P_\Theta\{\theta^k - \alpha g_{w^k}(\theta^k)\},$$

(3.12)

where $P_\Theta\{\cdot\}$ denotes projection on the set $\Theta$ and $\alpha > 0$ is a fixed stepsize. Instead of setting $w^k$, variables which indicates the node who does the upgrade at step $k$, as an IID taking values from the set $\{1, \ldots, N\}$ with equal probability, in [17] they let $w^k$ be the state of a Markov chain corresponding to the communication structure. In addition they assume that the Markov chain is irreducible, acyclic, and have the uniform distribution as its stationary distribution. This assumption means that the underlying communication structure is connected; after a sufficient amount of time the chain can be in any state; and that all state in the chain will be visited the same number of times in the long run. The way used in [17] (and that we use in [4.2.2]) to construct the transition matrix using only local information was the Metropolis-Hastings.

A Markov chain, named after Andrey Markov, is a discrete-time stochastic process with the Markov property. Having the Markov property means the next state solely depends on the present state and does not directly depend on the previous states.
scheme [36]. If the underlying communication topology is connected, then all assumptions on the transition matrix is fulfilled if the elements are set to
\[
P_{ij} = \begin{cases} 
\min\left\{ \frac{1}{d_i}, \frac{1}{d_j} \right\} & \text{if } (i, j) \in E \text{ and } i \neq j \\
\sum_{(i, k) \in E} \max\{0, \frac{1}{d_j} - \frac{1}{d_k}\} & \text{if } i = j \\
0 & \text{otherwise,}
\end{cases}
\] (3.13)
where \(d_i\) is node \(i\)'s number of edges.

Algorithm 3 Peer to peer unicast optimization algorithm
1: Initialize \(\theta^0\) and \(\alpha\). set \(k := 0\) and \(w^k := 1\).
2: repeat
3: At node \(w^k\), compute a subgradient, \(g_{w^k}\), for \(q_{w^k}(\theta^k)\).
4: \(\theta^{k+1} := \text{P}_\alpha\{\theta^k - \alpha g_{w^k}\}\)
5: Send \(\theta^{k+1}\) to a random neighbor, \(w^{k+1}\), with transition probability according to \(P\).
6: \(k := k + 1\)
7: until convergence.

In our case of calculating the average, the update function at node \(i\) is
\[
\theta^{k+1}_i = \theta^k_i - \alpha(\theta^k_i - x_i),
\] (3.14)
where \(x_i\) is the local measurement of node \(i\).

For the proof of the convergence we refer to [17]. In our implementation, we have a convergence between a bound, with an oscillatory behavior, and the best estimate is close to the optimal point.

3.3.2 Broadcast

For this algorithm, we had as basis the work of Xiao and Boyd [36]. There, the authors want to find optimal weights to have the fastest possible convergence in solving the problem of finding the average of the values of different nodes in a network, using distributed linear iterations, which have the form
\[
x^{k+1}_i = W_{ii}x^k_i + \sum_{j \in N_i} W_{ij}x^k_j, i = 1, \ldots, n,
\] (3.15)
where \(k = 0, 1, 2, \ldots\) is the iteration index, and \(W_{ij}\) is the weight on \(x_j\) at node \(i\). In [36], they prove that a necessary and sufficient condition for the (3.15) to converge, is
\[
\lim_{k \to \infty} W^k = \frac{11^T}{n}.
\] (3.16)
We report here the conditions for the (3.16) to hold. For the proof we refer to [36, section 2]. The equation (3.16) holds if and only if
\[ W_1 = 1, \]  
\[ \rho(W - 11^T/n) < 1, \]  
where \( \rho(\cdot) \) denotes the spectral radius of a matrix.

We give some interpretations of these equations (see [36, section 2]):

- Equation (3.17) states that \( 1 \) is a left eigenvector of \( W \) associated with the eigenvalue one. This condition implies that \( 1^T x_{k+1} = 1^T x_k \) for all \( k \), i.e., the sum (and therefore the average) of the vector of node values is preserved at each step.

- Equation (3.18) states that \( 1 \) is also a right eigenvector of \( W \) associated with the eigenvalue one. This condition means that \( 1 \) (or any vector with identical entries) is a fixed point of the linear iteration (3.15).

- Together with the first two conditions, condition (3.19) means that one is a simple eigenvalue of \( W \), and that all other eigenvalues are strictly less than one in magnitude.

- If the elements of \( W \) are nonnegative, then (3.17) and (3.18) state that \( W \) is doubly stochastic, and (3.19) states that the associated Markov chain is irreducible and aperiodic.

This work is closely related to the problem of finding the fastest mixing Markov chain on a graph, see [36], work used also in [17], basis for our implementation of the unicast algorithm described above. There was the opportunity of setting the weights exploiting again the Metropolis-Hastings scheme we already implemented for the unicast algorithm. Indeed, we set the weights as
\[ W_{ij} = \begin{cases} \min\{\frac{1}{d_i}, \frac{1}{d_j}\} & \text{if } (i,j) \in \mathcal{E} \text{ and } i \neq j \\ \sum_{(i,k) \in \mathcal{E}} \max\{0, \frac{1}{d_j} - \frac{1}{d_k}\} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases} \]  
where \( d_i \) is node \( i \)'s number of edges. This definition of the weights fulfills the conditions of convergence above, so we are sure to have convergence (indeed we have). As pointed out in [36, sections 4.2 and 5], this is not the fastest way to set the weights (even if it is not bad), but our purpose was to obtain convergence with a broadcast algorithm, and use the same structure (slightly modified) as in unicast algorithm was very useful in implementation.
In the paper by Xiao-Boyd [36], they do not make any assumptions about synchronization in the network, but if we look at the equation (3.15), we see that a kind of synchronization is required, because at each step, every node should have values from the previous step of the other nodes, without "jumping" some steps. This version of the broadcast algorithm we implemented, the reliable one, following this criteria, makes every node wait for data acknowledgments from all its neighbors before doing the update step, and similarly each node has to wait to retrieve data from all its neighbors before update. In this way we are sure to have synchronization for the steps in each node. In the next algorithm, unreliable broadcast, we try to remove these constraints.

Following we present here a code for the algorithm, supposing to be in the node i:

**Algorithm 4 Reliability Broadcast**

1: **In transmission:**
2: Transmit actual value to neighbors in broadcast
3: repeat
4: if backoff timer expires then
5: send again the value
6: end if
7: until all acks received
8: **In reception:**
9: repeat
10: if a packet arrives then
11: store value
12: send acknowledgment
13: end if
14: until data from all neighbors received
15: **Update step:**
16: repeat
17: if all acks received AND all packets received then
18: set the weights for each neighbor following the **Metropolis-Hasting** scheme
19: do the update step $x_i^{k+1} = W_{ii}x_i^k + \sum_{j \in N_i} W_{ij}x_j^k$
20: end if
21: until convergence
3.3.3 Unreliable broadcast

This algorithm is a modification of the previous algorithm. We wanted to investigate if it was possible to reach convergence, using the same update step as the reliable broadcast algorithm, but without any acknowledgments and synchronization. The problem was mainly to investigate whether it was possible to change the weights in the equation (3.15) in a way to take into account missing values, still having convergence. Finally, we found that simply setting to 0 the weights for neighbors with no new values, and therefore increasing the weight assigned to the local data of the node in which we are doing computation, we have convergence. So we can rewrite the *Metropolis-Hasting* scheme in this way:

\[
W_{ij} = \begin{cases} 
\min\left\{\frac{1}{d_i}, \frac{1}{d_j}\right\} & \text{if } (i, j) \in \mathcal{E}, i \neq j, j \text{ has sent a packet to } i \\
\sum_{(i,k) \in \mathcal{E}} \max\{0, \frac{1}{d_j} - \frac{1}{d_k}\} & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]

(3.21)

Here is the pseudocode of the algorithm:

**Algorithm 5 Unreliable Broadcast**

**In transmission:**

if transmission timer expire then

send actual value to neighbors, without waiting for response

end if

**In reception:**

if a new packet arrives then

store value

end if

**Update step:**

repeat

if Update timer expire then

set weights to neighbors whose there is a new packet, following the *Metropolis-Hasting* scheme and setting to 0 the value from neighbors whose packet is missing

do the update step \(x_i^{k+1} = W_{ii}x_i^k + \sum_{j \in \mathcal{N}_i} W_{ij}x_j^k\)

end if

until convergence

A problem of both reliable and unreliable broadcast algorithms, could be that we use the local sensor value only the first time each node send broadcast
3.3. ALGORITHMS AND NAMING

data, so, if this value is changing, it is difficult to recognize. A way to correct
this problem could be to take in parallel an old value and begin again the
iterations to see whether measurements are significantly changed.

3.3.4 Dual

For this algorithm, we have followed the work of M. Rabbat, R. Nowak, and
J. Bucklew in [26]. The purpose was to investigate whether the dual opti-
mization, without any kind of synchronization or acknowledgment, gives some
advantages over other methods.

We make the same assumptions as for the other methods described above: we
require a strongly connected graph and we use only local communications be-
tween neighbors. We provide here the theoretical basis behind this algorithm,
quoting for the most part the original work by Rabbat and Nowak [26].

In [26], they start defining a problem very similarly as for the other algo-
rithms we implemented,

\[
\text{minimize } \sum_{i=1}^{N} f_i(x),
\]

over \(x \in \mathbb{R}^d\), where the functions \(f_i(x)\) only depend on information at node
\(i\) and there are \(N\) nodes in the network (see also other works from the same
authors, like [25]). They then derive a primal-dual algorithm (see section
3.1.2), formulating a problem equivalent to (3.22) with constraints that impose
local communication and a consensual outcome:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{N} f_i(x_i) \\
\text{subject to} & \quad x_i = x \\
& \quad x_i \in \mathbb{R}^d,
\end{align*}
\]

where \(x_i\) is the local node \(i\) variable. The \(f_i\) are strictly convex, continuously
differentiable functions which are bounded below (but these assumption are
not critical here). In the formulation above, (3.23) the problem is that the
constraints introduce a coupling which would require every node to commu-
nicate with every other node in order to reach a solution, but this issue can
be overcome by rephrasing the problem in terms of local constraints:

\[
\begin{align*}
\text{minimize} & \quad \sum_{i=1}^{N} f_i(x_i) \\
\text{subject to} & \quad x_i - x_j \geq 0, \text{ for all } j \in \mathcal{N}_i, i = 1, \ldots, N.
\end{align*}
\]

The solution of this problem coincides with the solution of (3.22) provided
that the communication graph is strongly connected\(^6\).

\(^6\)because in order for the equality constraints to hold at all nodes in the network, there
is the requirement \(x_i \geq x_j\) and \(x_j \geq x_i\).
Next, Rabbat and Nowak decompose the problem (3.24) into $N$ subproblems which are local to each node. These subproblems have the form

$$\begin{align*}
\text{minimize} & \quad f_i(x_i) \\
\text{subject to} & \quad x_i - x_j \geq 0, j \in \mathcal{N}_i,
\end{align*}$$

where optimization is being performed on the variable $x_i$. They define a collection of Lagrange multipliers for the $i$th subproblem, $\lambda_i = \{\lambda_{i,j} : j \in \mathcal{N}_i\}$, and $x = \{x_i : i = 1, \ldots, N\}$, $\lambda = \{\lambda_i : i = 1, \ldots, N\}$. The Lagrangian function for the primal subproblem, (3.25), is

$$L_i(x_i, \lambda_i) = f_i(x_i) + \sum_{j \in \mathcal{N}_i} \lambda_{i,j} (x_i - x_j).$$

They define $q_i(\lambda_i) = \inf_{x_i} L_i(x_i, \lambda_i)$. Then the dual subproblem is

$$\max_{\lambda_i} q_i(\lambda_i),$$

over $\lambda_i$. The primal-dual approach to solving this problem alternates between iterates

$$\begin{align*}
x^k_i &= \arg \min_{x_i} L_i(x, \lambda^k) \\
\lambda_{i,j}^{k+1} &= \lambda_{i,j}^k + \mu (x^k_i - x^k_j),
\end{align*}$$

where $\mu > 0$ is a small constant step size (see references on [26]). This algorithm calculates the value of the dual cost function $q_i(\lambda_i)$ near the current iterate $\lambda^k_i$. Then they try to maximize $q_i(\lambda_i)$ to solve the dual problem. Based on the assumption that $f_i$ is strictly convex, it is possible to compute $x_i^{k+1}$ in the first step by solving $\frac{\partial}{\partial x_i} f_i(x_i) = -\sum_{j \in \mathcal{N}_i} \lambda_{i,j}^k$ for $x_i$. Then, to increase the dual cost function value, Rabbat and Nowak perform a gradient ascent step on the Lagrange multipliers $\lambda_{i,j}^k$.

The underlying principle behind this "primal-dual" approach is that, by solving the dual problem, the authors simultaneously solve the primal problem which they were originally interested in. More precisely, because the $f_i$ are strictly convex, we are guaranteed that there exists a solution $(x^*, \lambda^*)$ for which

$$\begin{align*}
\inf_x f_i(x) = f_i(x^*_i) = L_i(x^*, \lambda^*) = q_i(\lambda^*) = \sup_{\lambda} q_i(\lambda),
\end{align*}$$

and $\nabla_x L_i(x^*, \lambda^*) = 0$. The Lagrangian function for the network-wide problem (3.24) is $L(x, \lambda) = \sum_{i=1}^N L_i(x, \lambda)$. The condition $\nabla_x L_i(x^*, \lambda^*) = 0$ for each subproblem implies that $\nabla_x L(x^*, \lambda^*) = 0$; i.e., if $(x^*, \lambda^*)$ simultaneously solves the subproblems at each node then it also solves the network-wide problem.
3.4. ROBUST OPTIMIZATION

For the implementation details, see section 4.2.2. For the proof of convergence, we refer to [26].

In our case, see section 4.2.2, we have a network of sensors that are not synchronized. Each sensor upgrades its own value according to the equations above (in particular (3.28) and (3.29)) every $\zeta$ seconds, where $\zeta$ is a parameter, not correlated with the arrivals of new packet from other nodes. In reception, when a new packet arrives, the receiving node upgrades the value of $\lambda_{i,j}$ relative to the sender. We can present a pseudocode of the algorithm in this way, supposing to be in the node $i$ and supposing to have a timer that expires every $\zeta$ seconds:

**Algorithm 6 Dual broadcast distributed optimization algorithm**

1: In transmission:
2: repeat
3: if Timer expire then
4: $x_i^k = \arg\min_{x_i} \mathcal{L}_i(x, \lambda^k)$
5: end if
6: until convergence

7: In reception:
8: if a new packet arrives from node $j$ then
9: compute $\lambda_{i,j}^{k+1} = \lambda_{i,j}^k + \mu(x_i^k - x_j^k)$
10: end if

In the case of the average, the upgrade equation is

$$x_i^k = u_i - \sum_{j \in N_i} \lambda_{i,j}^k,$$

where $u_i$ is the scalar measurement of each sensor.

3.4 Robust Optimization

Another field of interest is about robust optimization, i.e., if some node, as often happens, has some problem or has not feasible data, it would be good to have less sensitivity to large measurement errors respect to least squares measures. A way to reduce sensitivity to outliers, could be to use the Huber Norm. This is a very interesting field, even if outside the scope of this work. We refer to the work and the references in [37] for details.
Chapter 4

Implementation
This implementation chapter wants to illustrate the tools used for this work in practical and wants to be a practical help to people who want to do similar things. We describe here also some code.

4.1 ns2

The most important tool for our work was the Network simulator 2 ns2. It is a discrete event driven simulator developed at UC Berkeley as a part of the VINT project. It is open source and versions are available for FreeBSD, Solaris, Windows, Linux, and Os X (the platform we used), and it is widely used, maintained and developed as a collaborative environment by a lot of institutes and people.

As showed in the manual, ns is an object oriented simulator written in C++, with an OTcl (Object Tool Command language) interpreter as interface. This duality gives big advantages to the user, because it is possible to exploit the powerful, fast execution time and efficiency of the C++ language to write the cpu-intensive ”core” of the code (e.g protocols), and in the same time it is possible to have the simplicity of the OTcl scripting language, for quickly change different parameters without the need of recompiling. So, two code hierarchies are supported, the C++ one (compiled) and the OTcl one (interpreted), with one to one correspondence between them.

4.1.1 Structure and functioning of ns2

Ns2 has a very modular structure (see figure 4.1 for an overview of the directory structure of ns2). All the objects are derived from the class TclObject and are first instantiated in the OTcl interface. The OTcl can make use of the objects compiled in C++ through an OTcl linkage that creates a matching of OTcl object for each of the C++.Ns2 is in practice a glue between Tcl and C++. It is a discrete event simulator, where the advance of time depends in the timing of events which are maintained by a scheduler. An event is an object in the C++ hierarchy with an unique ID, a scheduled time and the pointer to an object that handles the event. The scheduler keeps an ordered data structure with the events to be execute and fires them one by one, invoking the handler of the event (see section 1.1)).
4.1. NS2

An ns2 simulation consists of communications between objects. Every link, node, channel model, propagation model, application and the simulator instance itself is an object and the powerful thing is that it is possible to combine all that objects in almost any way, following our needs [32, 24, 6]. At the beginning we have to define a topology, i.e., where our nodes are in the environment. In wireless simulations it is also possible to define node movement. After defining the topology we have to define the features that each node should use in our simulation, like the physical layer, the MAC layer, the link layer, the routing layer, the type of interface priority queue (IFQ) (i.e., the buffer at the entrance of the node) and its maximum length, the type of antenna and the propagation model (for wireless). See [32, chap 5 and 16] for the list of all the possibilities. Then, we configure the agents we want to use in our simulation, i.e., the transport layer and application layer objects. We do all these things, and more, from the Tcl interface, so if we do not need to modify C++ code and we want only to do tests on already implemented protocols, we can do it very quickly, without recompiling ns2 and writing merely some simple (more or less) Tcl script.

When, as in this thesis work, we have to simulate something not already implemented, we can write C++ code and add our code to the general structure of ns2, recompile all, and use our object in simulations. Of course it is also possible to modify existing code, as ns2 is an open-source project. The best starting point to learn how the simulator work is indeed to try to modify existing code to change protocols behavior.

Installing

Installation of single packages of ns2 is quite tricky, in particular on Windows, where it needs the Cygwin package to work (www.cygwin.com), but the installation process is a lot easier on all platforms using the all-in-one package [24].

On OsX we installed ns2 without problems, just executing the 'install' script and setting the right environment paths as suggested by the installation’s script. For using NAM, the network animator (see next section), we need the X11 package that is present in the MacOsX’s dvd or can be easily downloaded from the apple website.

For recompiling ns2, we need gcc in our system. With MacOsX it is very easy: the only thing to do is to install the Apple’s Xcode package that we can find in the MacOsX’s dvd. Also in almost all Linux distributions gcc is already present or it can be installed easily. In cygwin we have to select the gcc package during cygwin’s installation process.
We need to recompile ns2 if we change something in the C++ code or if we write new code. If we already have gcc installed, to recompile ns2 (independently from which platform we are working in) we just need to type the commands ‘make depend’ and ‘make’ in the main ns2 directory. This process usually is not so long, because every time only the modified parts will be recompiled.

### Output and tools

We can define three types of outputs for ns2:

- Trace output
- Nam output
- Log output

The trace output is the main ns2 output, there it is possible to find all the packets sent, received and dropped in the simulation, with informations about address of the sender and recipient, energetic informations and different informations for each protocols and for different types of packets. If we write, as we did in our work, some new code, it is possible to put our particular output in the trace file, modifying (for wireless simulations) the files `trace/cmu-trace.h` and `trace/cmu-trace.cc`.
4.1. NS2

Figure 4.2: A five-nodes wireless configuration in the nam window exchanging broadcast and unicast data

Nam, the *network animator*, is, as we can find on the introduction of chapter 45 in [32],

a Tcl/TK based animation tool for viewing network simulation traces and real world packet tracedata. The design theory behind nam was to create an animator that is able to read large animation data sets and be extensible enough so that it could be used indifferent network visualization situations. (. . . ) When the nam trace file is generated from ns2 or others applications, it is ready to be animated by nam. Upon startup, nam will read the tracefile, create topology, pop up a window, do layout if necessary, and then pause at time 0. (. . . ) Through its user interface, nam provides control over many aspects of animation.

The nam animations are very useful to give a general idea of how the network works. There are (see figure 4.2) commands for play, play in reverse, forward and rewind the animation and for setting the animation’s speed. Broadcast packets are represented as circle, while unicast packets are represented by dots. It is possible to have info about each packet, link or node, just clicking on it.

The log output is the output we have from our C++ code (it could be done in several ways, e.g with *fprintf(stdout,"Message we want to print");*)
and from the Tcl script (with the \texttt{puts} command). In this way it is possible to print useful information that is not possible to put in trace files. The destination of these informations is the standard output but we can redirect to a file with the ‘\$’ operator.

For parsing the different outputs it is better to use external applications, like \texttt{awk} or \texttt{perl} (see introduction to results section for details).

For create and editing the code under MacOs X we used the application \textit{Ko-modo edit}. \cite{3}.

\subsection{802.15.4 in ns2}

The 802.15.4 protocol is implemented in ns2 thanks to J. Zheng and M. J. Lee, \cite{38, 40}. In figure \ref{fig:ns2 Simmons} we present the function modules in the simulator and the services implemented in the PHY and MAC modules (see also section \ref{sec:phy} and \ref{sec:mac}). Unfortunately no real documentation exists on the implementation, but it seems that, even if there is some minor bug, the behavior of the simulator respect faithfully the standard IEEE 802.15.4, in the version P802.15.4/D18 Draft. In figure \ref{fig:ns2 Simmons} we show the general structure of the module and the service implemented in the PHY and MAC layers.
Flow of packets inside ns2

Here we show the path that each packet go through inside the ns2 structure in our case, in particular inside the module 802.15.4, developed by J. Zheng and M. J. Lee (see section 4.1.2). The module 802.15.4 is not so well documented, so we deduced the following informations from the code and from the work of Iyappan Ramachandran, [28].

Outgoing:

- The application we developed, ccsensorApp, sends a request to send (broadcast or unicast) to the agent ccsensorAgent, through CcsensorApp::sendSensorData() or CcsensorApp::sendDataUnicast().

- ccsensorAgent sends through Agent::send(packet ∗p, Handler ∗h) the packet to the classifier that, based on the field of the header, send it to the right module (to the routing protocol in the case of an outgoing packet)

- so classifier sends the packet to dump agent, a basic routing protocol (we do not use a routing protocol, but ns2 structure needs something at network layer, so there is this dump agent)

- the routing protocol sends the packet to the LLC sublayer of the data link layer, whose is part also the MAC sublayer. The function of LLC sublayer here is only to deliver the packet to the arp and to the queue it

- In the arp we translate the ip address into MAC address. We deactivated the arp module because without the routing protocol, we did not need address resolution, and we did not want to take in consideration arp packets.

- queue is in this case the typical Drop Tail, a FIFO scheduling and drop-on-overflow buffer management.

Now we describe in a bit deeper detail what happen inside the 802.14.4 module:

- The upper layer (queue) hands down a packet to MAC by calling MAC802.15.4::recv(Packet ∗P, Handler ∗h).

- recv() then calls Mac802.15.4::mcps_data_request(), which works giving control to other functions, selected basing upon a variable called step, initialized to 0 and incremented before passing control to other functions.
In case of direct transmission, `mcps_data_request()` calls `Max802_15_4::csmacaBegin(pktType)` after incrementing step to 1.

- `csmacaBegin` calls `CsmaCA802_15_4::start()`. `start()` calculates the random backoff time, determines if it can proceed and starts the macBackoffTimer with the backoff time determined.

- When the timer expires, `CsmaCA802_15_4::backoffHandler()` is called. `backoffHandler` then turns on the receiver (calling `Phy802_15_4::PLME_SET_TRX_STATE_request()`), and requests a CCA by calling `Phy802_15_4::PLME_CCA_request()`.

- CCA, is done starting the CCAH timer, with 8 symbol durations, which on expiry calls `Phy802_15_4::CCAHandler`. `CCAHandler` determines if the channel is idle and reports the answer by calling `Mac802_15_4::PLME_CCA_confirm()`.

- If the channel is idle and no collisions are detected, `PLME_CCA_confirm()` calls `backoffHandler` to perform CCA again. If the channel is found busy, `CsmaCA802_15_4::start()` is called again.

- Now the control go back to `Mac802_15_4::mcps_data_request()` which enables the transmitter by calling `Phy802_15_4::PLME_SET_TRX_STATE_request()`, after incrementing step to 2.

- `Mac802_15_4::PLME_SET_TRX_STATE_confirm()` passes the data to `MAC802_15_4::txBcnCmdDataHandler()`, which gives the data to `Phy802_15_4::recv(Packet *P, Handler *h)`.

- `recv()` calls `Phy802_15_4::PD_DATA_request()`, which uses `WirelessPhy::sendDown()` to transmit the data to `Channel::recv()`.

**Incoming:**

- `Channel::recv()` gives one copy of the packet to each node using `WirelessChannel::sendUp()`, which subsequently passes the packet to `Phy802_15_4::recv()` after propagation delay.

- `Phy802_15_4::recv()` indicates packet reception to MAC using `Phy802_15_4::PD_DATA_indication()`, which calls `Mac802_15_4::recv(Packet *P, Handler *h)`.
4.1. NS2

- recv() drops the packet if there is a collision or calls Mac802_15_4::recvData() if there is no collision. Data are passed to the upper layer by Mac802_15_4::MCPA_DATA_indication().

- The path in incoming direction through the upper layers is very similar (in reverse way) to the path in outgoing direction described above, but simpler.

Changes to ns2 structure

To create a new agent in ns2 and a new packet type, we have to modify also some files in the ns2 structure (see chapter 10 of [32], [29] and [2]). In our case, after defining the agent (see section 4.1.3), we had to inform ns2 about the type and the name of the packet sent by our agent and our ADU (see section 4.1.3). This is the list of the files we modified:

- in /common/packet.h, in the enum packet_t section, we put the name of our packet, PT_CCSENS, and in the class p_info we put name _[PT_CCSENS]="ccsens"

- in /common/ns-process.h, we added definition for our ADU, CCSENS_DATA, in the enum APPDataType structure

- in /tcl/lib/ns-packet.tcl, we added our packet name, Ccsens, in the foreach prot structure.

- in /trace/cmu-trace.cc, we created a function to write informations about our packet in the trace file. We added the function definition, format_ccsens(p, offset), in the nested switch structure in the function CMUTrace::format(Packet* p, const char *why). Of course we had also to put the function definition in the /trace/cmu-trace.h.

- in /tcl/lib/ns-default.tcl we had to add the default values of our bind variables (see section 4.1.4): Application/CcsensorApp set actualValue_ 0; Application/CcsensorApp set nodeData_ 0; Application/CcsensorApp set alpha_ 0.01; Application/CcsensorApp set bindTime_ 1; Application/CcsensorApp set mu_ 0.01

4.1.3 New agent for ns2

In our work, we had to create an application, so we needed an agent to deal with application’s data.

---

2 all file paths are referred to the ns2 root directory.
Problems and reason for a new agent

What we wanted to do in our work was to implement in ns2 different algorithms for distributed optimization in sensor networks, using only communications between neighbors (i.e. without multi-hop passages), see [3.1]. For doing our purposes, it was essential to exchange scalar values between neighbors. In ns2 there are some already implemented applications, like telnet and FTP (see chapters 38-39 of [32]), but they are 'virtual', the all they transmit is only size and time when data are transferred. For sending real application data, ns2 provides a packet-like structure, called ADU (chapter 39.1.1 [32]), to pack user data into an array, which is then included in the user data area of an ns2 packet by an Agent, but there is not support yet in the class agent to transmit user data, so there were two possibilities: use the tcpApp application (a kind of wrapper application for sending user data) or build a new agent and send data through it.

We tried both ways, and finally the best solution was to write a new agent to have more flexibility. With TcpApp it was indeed difficult to send broadcast packets, even if at the beginning it is easier to modify already existent code than write new one. After writing our agent, ccsensorAgent, it was possible to built our application, ccsensorApp, to implement the algorithms. A good start point for writing new protocols and agents is [29] and chapter 10 of [32].

We did not use a routing protocol because we need to send data only to neighbors, and for the same reason we disabled also the arp protocol (for neighbors we can use the MAC address, we do not need arp translation).

Adu definition

First, in the file ccsensor-packet.h, we defined our ADU, derived from the abstract base class of all ADU, AppData:

```
file: ccsens/ ccsens-packet.h

class CCSensData : public AppData {
private:
unsigned char code_;
unsigned char iterNumber_;
double data_;

public:
CCSensData();
CCSensData(char * b);
```
4.1. NS2

Each ADU contains:

- a code for differentiating the type of message we want to send
- an iteration number to recognize new packets
- data we want to send

Then are defined the functions to access directly data and to pack in an array. In an ADU should be always provided at least the constructor, to extract data from an array and the `pack` function, to pack data into an array.

We have to provide also an header for accessing our ADU from the agent and from the application:

file: /ccsens/ccsens-packet.h

```c
struct hdr_ccsens{
    unsigned char code_;  
    unsigned char iterNumber_; 
    double data_; 

double& data() { return (data_); } };
void setHeader(hdr_ccsens *ih);

virtual int size() const { return sizeof(CCSensData); }  
virtual AppData* copy() { return (new CCSensData(*this)); }  
void pack(char* buf) const;
void print();
};
```

```c
Each ADU contains:

- a code for differentiating the type of message we want to send
- an iteration number to recognize new packets
- data we want to send

Then are defined the functions to access directly data and to pack in an array. In an ADU should be always provided at least the constructor, to extract data from an array and the `pack` function, to pack data into an array.

We have to provide also an header for accessing our ADU from the agent and from the application:

file: /ccsens/ccsens-packet.h

```
Function of the agent

The functions of the agent are:

- Prepare the packet for broadcast message
- Prepare the packet for unicast message
- Receive new packets and give them to the application

The first two functions consist in setting the right values on the common header and on the ip header, and to send down the packet to the next step in the ns2 chain, in this case the logical link layer. For our application’s details see section 4.2.

To compile our files, we had also to modify the makefile, adding our files to compile under the line OBJECT_CC.
4.1.4 OTcl script and simulation run

The simulation itself and the main application where the algorithms are implemented (see section 4.2) are controlled by an OTcl script (figure 4.4) that defines which feature to use, and which part of the ns2 structure utilize. As we said in section 4.1.1, one of the most powerful features of ns2 is the complete correspondence and parallelism between OTcl and C++, to exploit advantages from both languages. In the OTcl script, we define, in the wireless case, characteristics like the topology we want to use, the number of nodes, the type of channel, the type of antenna, the network interface, the mac type, the propagation model and others (for a detailed list in general see [32, chapter 16], for a detailed list of the ones we used, see later in this section). Here we provide a detailed description of the OTcl script we wrote for our simulation. For reference about OTcl we found very useful the tutorial at [35].

Script description

At the beginning of the script, it is necessary to list and initialize all the variables we will use in the script, so to have a general view of the configuration we want and so to be able to quickly change parameters.

```otcl
#=================================================================================
# Define options
#=================================================================================
set val(chan) Channel/WirelessChannel ;#channel type
set val(prop) Propagation/TwoRayGround ;#radio-propagation model
```
set val(netif) Phy/WirelessPhy/802_15_4; #network interface type
set val(mac) Mac/802_15_4; #MAC type
set val(ifq) Queue/DropTail/PriQueue; #interface queue type
set val(ll) LL; #link layer type
set val(ant) Antenna/OmniAntenna; #antenna model
set val(ifqlen) 25000; #max packet ifq
set val(nn) 5; #number of mobile nodes
set val(rp) DumbAgent; #routing protocol
set val(x) 100; #horizontal size of topology
set val(y) 100; #vertical size of topology
set val(stop) 100; #duration of simulation
set val(cp) "wpanBasic3.scn"; #default scenario file
set val(tr) "simple.tr"; #default trace file
set val(nam) "simple.nam"; #default nam file
set val(errConv) "errConv.conv"; #default time/error output file
set val(errNode) "errNode.ncon"; #default node/error output file
set val(timePacket) "timePacket.out"
set val(energymodel) EnergyModel; #definition of energy model
set val(initialenergy) 100; #Initial energy in Joules
set val(brdcst) yes; #set which algorithm to use
set val(autoDiscovery) no; #automatic neighbor discovery
set val(reliable) yes; #choose reliable or unreliable broadcast
set val(dual) yes; #for setting dual or primal algorithm
set val(bindTime_) 1; #time for upgrade window in dual algorithm
set val(mu_) 0.1; #constant stepsize for dual broadcast algorithm
set val(alpha_) 0.01; #constant step size for unicast algorithm
set val(step) 0.01; #could be mu or alpha
set val(macAck) off; #set mac acknowledgment
set val(algorithm) "broadcast"; #only for naming output files

It is possible to change from command line several of this parameters, since the OTcl stores command line arguments in the variables argc (number of arguments) and argv (list of arguments). We do not provide here the simple OTcl function to do this because it is not strictly correlated with our thesis but only with OTcl programming. For that a very good tutorial could be found at [35]. We use some of the parameters defined above to define the features that each node should have:

$ns_ node-config -adhocRouting $val(rp) \n-adllType $val(ll) \n-macType $val(mac) \n
```nmandrake
-ifoType $val(ifq) \
-ifoLen $val(ifqlen) \
-antType $val(ant) \
-propType $val(prop) \
-phyType $val(netif) \
-topoInstance $topo \
-agentTrace ON \n-routerTrace OFF \n-macTrace ON \n-movementTrace OFF \n-channel $chan_1_ \n-energyModel $val(energymodel) \n-rxPower 0.3 \n-txPower 0.6 \n-initialEnergy $val(initialenergy)

and finally we create the nodes:

for {set i 0} {$i < $val(nn) } {incr i} {

global rng sum

set node_($i) [$ns_ node]
$node_($i) random-motion 0 ;#disable random motion
$god_ new_node $node_($i);#inform god (for neighborDiscovery)

set nodeData($i) [$rng uniform 0 4]

set b_($i) [new Agent/Ccsensor]
$node_($i) attach $b_($i) 250
Application/CcsensorApp set nodeData_ $nodeData($i);#sensor value
Application/CcsensorApp set bindTime_ $val(bindTime_)
Application/CcsensorApp set mu_ $val(mu_)
puts "local data node $i: $nodeData($i)"
set app_($i) [new Application/CcsensorApp]
$app_($i) attach-agent $b_($i)
$app_($i) broadcast $val(brdcst)
$app_($i) neighDiscovery $val(autoDiscovery);#automatic neighbors discovery
$app_($i) reliable $val(reliable);#use reliable broadcast or not
```
\$\text{app\_}($i)\ \text{dual} \ \$\text{val}(\text{dual}) \ ;\ #\text{dual or primal optimization}

set sum [expr $\text{sum}\+$\text{nodeData($i)\}]

\$\text{ns\_} \ \text{at} \ 0.1 \ "$\text{app\_}($i) \ \text{start}"

}

We can see that we also can give instructions directly to our application (see section 4.2) through the \textit{command} function and through \textit{variable bindings} between OTcl and C++. Then we start the simulation with the command \texttt{\$\text{ns\_ run.}} The script is more complex but here we wanted to show only the main aspects that should have a OTcl script for an ns simulation. We refer to OTcl tutorial [35] and to the ns manual [32] for further details.

\textbf{Variable connection between OTcl and C++}

OTcl and C++ can access and modify the same variables, and it is possible also to share results between the two structure (the OTcl hierarchy and the C++ hierarchy) and to give commands to C++ from OTcl and vice-versa. For a complete discussion about this very important and useful feature, that we used a lot in our work, see chapter 3 of [32]. Variables accessible from both structures are called \textit{bind variables}, e.g in our case \texttt{bindTime, nodeData, mu, and alpha} are bind variables in our application. It is better to put a default value for each bind variables in the file \texttt{ns-default}, as suggested by a warning (if the bind variable is not properly initialized) each time we execute our simulation.

To give direct command to an application, we have to implement the command in the \textit{command} function in the C++ code (see section 4.2). The way for execute OTcl command from C++ is trickier: we have first to access to the Tcl instance, \texttt{Tcl\& tcl =Tcl::instance();}, then evaluate our command (a string) with \texttt{tcl.eval(command);} and finally access to the result of the command, through \texttt{const char* ref =tcl.result().} Now we have the result in the const char variable \texttt{ref}, and we can use it for our purposes.
4.2 Our Application: CCsensorApp

Most part of our work was focused on programming the application CCsensorApp. Here are implemented in practice the algorithms we have worked on. What we wanted to do was a flexible structure to simulate different distributed optimization algorithms that use broadcast and unicast communications only between neighbors.

We use the same general structure to simulate different algorithms, having the possibility to choose which algorithm to use from the OTcl interface.

4.2.1 CCsensorApp’s general structure

In the file ccsensor/ccsensorapp.h, other than several timers, we defined two main classes, the class dataAddress, a container for data originated from each neighbor, and the application’s main class, ccensorApp.

The informations that we need from our neighbors, defined in the dataAddress class, are:

- address of the neighbor
- a buffer to store two different data values
- the number of neighbors of our neighbor
- the probability of send a packet to that node in the unicast optimization algorithm
- iteration number value, to identify if the packet that is arriving is a new one
- a flag to know if the value from that neighbors is a new one
- a flag to know if we have acknowledgment for our last iteration from that neighbor
- a value to know the number of the last iteration received from that neighbor
- the lambda value for our neighbor

Not all algorithms implemented use all fields of the dataAddress class, e.g the lambda value is used only in the dual broadcast optimization algorithm (see section [3.3.4]). In the main class, ccensorApp, we have a vector of dataAddress objects big enough to store information about all neighbors for each
node.

The main class is the one who implements the algorithms. It provides functions to send sensor data in broadcast (sendSensorData) and in unicast (sendDataUnicast); to process (process_data) packets arrived from neighbors through the underline structure; to do the upgrade step, different for each algorithm (upgradeValue). It is also implemented a function to do automatic discovery of neighbors (see section [4.2.3]), but it can be deactivated from Otcl interface for statistical purposes (we are not interested in packets sent for discovering the network, only in packets inherent in the algorithms’ data transfer). If the automatic discovery of neighbors is deactivated, each node knows who its neighbors are through god (General Operation Director), an ns object that has a global view of the current simulation and can easily calculate each node’s neighbors 3. The god object has to know the range of radio communication to work properly. We had to modify the file mobile/god.cc to make possible to set the radio range from OTcl interface.

Here we provide the complete list of functions in the CcsensorApp class, from the file ccsensorApp.h:

```c
file ccsensorApp.h

public:
 CcsensorApp();

 void process_data(char *d, hdr_ip* ih);

protected:
 int command(int argc, const char*const* argv);

void start(); // Start sending data packets
void stop(); // Stop sending data packets

private:
 void sendSensorData();
 void sendDataUnicast();
 void upgradeValue();
 void neighborsDiscovery();
 int getEdge(int neigh);
 void reset_sens_timer();
 ns_addr_t markov();

inline bool wait(){return neighWaiting;}
```

3See chapter 16 of [32], but the role of god is not so well discussed there. For more information, very good websites are [12] and [33].
4.2. OUR APPLICATION: CCSENSORAPP

```c
inline double randomDelay();
inline double randomDelay(double del);
```

Probably the most important function is the `process_data` one, which implements what to do when a packet arrives, considering its code number.

### 4.2.2 Implementation of the algorithms

The purpose of our thesis is to compare performances of different distributed algorithms to calculate average consensus in sensor networks. Here we will describe the way of implementation. For a mathematical description of the algorithms, see section (3.1).

The algorithms we have implemented are:

- Distributed subgradient algorithm with transmissions between neighbors following a Markov chain (matrix constructed using the Metropolis-Hasting scheme(see [17, 7])) (see also section 3.3.1)
- Reliable broadcast algorithm (see also section 3.3.2)
- Unreliable broadcast algorithm (see also section 3.3.3)
- Dual optimization (see also section 3.3.4)

We will discuss and compare results in simulations in section 5.2.

**Distributed sub-gradient algorithm with transmissions between neighbors following a Markov chain**

We select this configuration from the OTcl script, setting the variable `val(brdcst)` to ‘no’ in our Tcl script (section ). For this algorithm (see figure 4.5) to work, it is required that is one and only one node who begins transmissions. In our simulation we let the node 0 start from OTcl, but in a real network it would be an interesting problem to establish which node begins⁴.

To make this algorithm work properly, we should also turn on the MAC layer acknowledgment of IEEE 802.15.4. We can do this from OTcl, setting `Mac/802_15_4 wpanCmd ack4data` to on. The MAC acknowledgment does not follow the CSMA/CA mechanism but it is sent immediately in the channel.

---

⁴Possible solutions are beyond the purposes of this work, but we could think to some random timer to begin a transmission if no packets are received before the timer expire. This solution could also avoid a failure of a node.
In this algorithm, as we said in the math section, each node receives a value, does the upgrade step, and sends the new value to a neighbor, following the state of a Markov chain. The steps inside our application at node $i$ are:

1. wait for finish neighbors discovery, if it is activated (see section 4.2.3)
2. if this node begins: schedule transmission of data (the value of the sensor the first time) through the \textit{sendDataUnicast} function.
3. wait to receive a packet
4. when a packet arrives (function \textit{process\_data}, code 1) store the data value in the apposite vector, following the source address
5. do the upgrade step in the \textit{upgradeValue} function, $	heta_{i}^{k+1} = \theta_{i}^{k} - \alpha \times (\theta_{i}^{k} - x_{i})$, where $\theta_{i}^{k}$ is the node value and $x_{i}$ is the sensor value (see section 3.1)
6. transmit new value to a neighbor selected through a Markov chain

\textbf{How the Markov function works:} Each time we want to send data with the unicast algorithm, we select a neighbor with the \textit{markov} function. Our is a special Markov chain that takes on values in the finite set of the neighbors of each node, with transition matrix $P$ (see [17] and section 3.1). To build the transition matrix we used the Metropolis-Hasting scheme ([17, 7]), with the elements set to:

![Diagram of the unicast algorithm](image-url)
4.2. OUR APPLICATION: CCSENSORAPP

\[
P_{ij} = \begin{cases} 
\min\left(\frac{1}{d_i}, \frac{1}{d_j}\right) & \text{if } (i, j) \in \varepsilon \text{ and } i \neq j \\
\sum_{(i,k)\in\varepsilon} \max\left\{0, \frac{1}{d_i} - \frac{1}{d_k}\right\} & \text{if } i = j \\
0 & \text{otherwise}
\end{cases}
\]

where \(d_i\) and \(d_j\) are the number of neighbors of the node \(i\) and \(j\).

When we execute the markov function, we are in a particular node, so we have knowledge only about the \(i\) row of the matrix, therefore the probability to transmit to our neighbor \(j\) is the minimum between our number of neighbors and the number of neighbors of node \(j\), that we have in the dataAddress structure (see section 4.2.1).

For selecting a node using its transition probability, we ordered our list of neighbors in increasing order and we select one of them with the help of a random value (see chap 25 of [32] for random number generation in ns2). The markov function return finally the address of the selected neighbor. If no neighbors are selected we do another upgrade step in the same node (with probability \(1 - \sum_j P_{ij}\)).

For convergence statistics and results see the section 5.2.

Reliable broadcast algorithm

We select this configuration from the OTcl script, setting the variable \(\text{val(brdcst)}\) to 'yes' in our Tcl script (section 4.1.4). We can turn off the MAC acknowledgment because we already send acknowledgments at application level. This will give us more realistic simulations and let us avoid too many useless collisions. In this algorithm, each node send a broadcast message to its neighbors with its local value, and wait for acknowledgment from all of its neighbors. It also waits to have all data from all its neighbors. If both conditions are fulfilled, the node has received all acks for its message and it has all value from its neighbors, it does the upgrade step. So here we have a kind of synchronization between the nodes, because each one has to wait for the others to be ready.

For this purpose, it is also important to keep track of the sequence number of each packet, so it is possible to recognize if the packet arrived is a new one or an old one. It could happen that a node do the upgrade step while some of its neighbors are still waiting for others, so they receive a new value without have used the old one. For this purpose we have a buffer to temporarily store new data to use in the next iteration. For each packet arrived, even if it is an old packet, each node send an acknowledgment.

Here the upgrade step for the node \(i\) is (see also section 3.1): \(\text{newValue} = (\sum_{j\in\mathcal{N}_i} W_{ij} \times \text{data}(j)) + W_{ii} \times \text{data}(i)\) where the weight for each neighbor (ma-
trix $W$) is given again using the Metropolis-Hasting scheme (see also [36]):

$$W_{ij} = \begin{cases} \min(\frac{1}{d_i}, \frac{1}{d_j}) & \text{if } \mathbf{ij} \in \mathcal{E} \text{ and } i \neq j \\ \sum_{(i,k) \in \mathcal{E}} \max\{0, \frac{1}{d_i} - \frac{1}{d_k}\} & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

In detail, the steps inside our application are:

In transmission:

1. wait for finish neighbors discovery, if it is activated (see section 4.2.3)
2. wait for a random delay$^5$ and send data in broadcast (function `sendSensorData`)
3. wait for acknowledgments from all neighbors and eventually send data again a MAXATTEMPTS number of time. The time to wait was empirically established.
4. when all acks are received, put to true the flag `okUpgrade` and cancel backoff timer for send data again

In reception:

1. when a packet arrives, it is processed by the function `process_data` (code 3 for a reliable broadcast packet).
2. if it is a new packet (looking at the iteration number), store data in the data field of the `dataAddress` structure for the appropriate neighbor
3. if it is a new packet, but the previous one was not yet used in the upgrade step, store the new data in the `dataBuffer` field of the `dataAddress` structure for the appropriate neighbor
4. send a reply for the packet received (no waiting at application level, but `CSMA/CA` at MAC layer (see [15]))
5. check whether there are new data packets from all neighbors. If there are, check if the `okUpgrade` flag is true (all acks received for the last packet sent) and if it is, to the upgrade step
6. if the `okUpgrade` flag is false, schedule the Upgrade Timer to wait and try again when the timer expires (interval set empirically)

$^5$it is possible to avoid this because there is already a random delay at mac layer with `CSMA/CA`(see [15] for detailed explanations about csma/ca in 802.15.4).
4.2. **OUR APPLICATION: CCSENSORAPP**

For the upgrade step:

1. execute the *markov* function to assign the appropriate weight to each neighbor
2. assign to the current node a weight equal to $1 - \sum_{j \in N_i} W_{ij}$
3. do the upgrade step (see above) $newValue = (\sum_{j \in N_i} W_{ij} \times data(j)) + W_{ii} \times data(i)$
4. update the iteration number
5. check the buffer if there are already new packets
6. send a broadcast message with new data (through the function `sendSensorData`)

This algorithm has a very fast convergence to the optimum value. Problems are about the number of packets and the thing that we use the local sensor value only for the first packet sent, so it is more difficult to recognize if the sensor data changes (see section 5.2).

**Unreliable broadcast algorithm**

This algorithm is quite similar to the reliable broadcast one, but it does not have any type of acknowledgment and synchronization. The update function is the same of the reliable broadcast case, but is assigned a weight only to the nodes whose is present a new data packet. To the others we assign a weight of zero. In practice if we are doing the upgrade step in the node $i$, following the formula $W_{ii} = \sum_{j} W_{ij}$ for $i = j$, we assign a bigger weight to the value of the node $i$ itself. The steps inside our application are:

1. wait for finish neighbors discovery, if it is activated (see section 4.2.3)
2. wait a random delay$^5$ and send data in broadcast (function `sendSensorData`)
3. if the backoff time is not stopped, continue sending again the data when the backoff timer expires

In reception:

1. when a packet arrives, if it is a new packet, store data as in reliable broadcast and set the flag *validate* for the node who sent the packet
CHAPTER 4. IMPLEMENTATION

Figure 4.6: Reliable broadcast algorithm
2. if it is not started, start the timer to do the upgrade step (interval settable through OTcl script)

3. when the timer expires, do the update step as in reliable broadcast but counting only the neighbors from who sent a new packet

4. set the timer for the next upgrade step

The only parameter is the interval for upgrading. The problem is still that we use the sensor value only at the beginning.

**Dual optimization**

We select this configuration from the OTcl script, setting the variable `val(dual)` to yes in our Tcl script (section...) It does not matter whether the mac acknowledgment is set on or off, because we only send broadcast packets here, and there are never acknowledgments for broadcast packets at mac layer (see [15] for details about acks in 802.15.4). The aim of this algorithm (see math section and [26]) is to have a reliable and sharp enough final value in all nodes (the average in our case), without any kind of synchronization and without any acknowledgments. Here each node do the update step every interval of time (settable from OTcl and not synchronized with other nodes), based on the values of the $\lambda_{i,j}$, a collection of Lagrange multipliers for the $i$th subproblem ($\lambda_i = \{\lambda_{i,j} : j \in \mathcal{N}_i\}$). If we are in the node $i$ and a packet arrives from the node $j$, we only upgrade the $\lambda_{i,j}$ value, not the local value of $i$. We upgrade the local value of $i$ after the upgrade interval, in a way not correlated with the arrive of packets or to what other nodes do (for mathematical details, see the relative section in this Thesis and [26]).The steps inside our application are:

**In transmission:**

1. wait for finish neighbors discovery, if it is activated (see section [4.2.3])
2. wait a random delay and send local value in broadcast
3. schedule the upgrade step after the interval set from OTcl
4. when the timer for the upgrade expires, do the upgrade step. For node $i$: $newValue = localsensorData - \sum_j \lambda_{i,j}$
5. send the new value in broadcast. We decide to send the value several times before the next update, to have more possibilities to reach a greater number of neighbors

**In reception:**
1. when a dual broadcast packet arrives (code 2), process in the process_data function

2. if it is a new packet (looking at the iteration number), store data in the data field of the dataAddress structure for the appropriate neighbor

3. if we are in the node $i$ and the source is the node $j$, we upgrade the $\lambda_{i,j}$ value following the formula $\lambda_{i,j}^{k+1} = \lambda_{i,j}^k + \mu \times (x_i - x_j)$, where $\mu$ is a fixed step size (settable from OTcl interface), and $x_i$ and $x_j$ are the local data respectively of the node $i$ and of the node $j$

In practice we are able to have convergence without any acks or synchronizations, even if convergence is usually slower than in the other algorithms (see section 5.2). We have a very good behavior changing the parameters of the upgrade interval and $\mu$. For results, considerations and comparison with the other algorithms, see the section 5.2 on this thesis.

### 4.2.3 Neighbors’ discovery

Beyond the scope of this thesis but very interesting problem for future work. In the application shown in this thesis there is already a function for automatic neighbors discovery: each node at the beginning of the simulation and after a fixable period (to recognize whether the topology is changed), sends in broadcast an hello packet. All nodes who reply, are considered neighbors. This approach seems to work well with a small number of nodes, but there are some problems with many nodes (depending from the topology of the network), because there are too many packets in the network with a lot of collisions, and therefore it takes a long time to discover all neighbors. However this approach seems good but needs to be improved. In the simulations the automatic neighbors discovery is disabled: each node knows how many neighbors it has by the already cited structure god.
Chapter 5

Results and Comments
CHAPTER 5. RESULTS AND COMMENTS

5.1 Types of comparison

To compare different algorithms, we had to choose some common metrics. For each algorithm (Broadcast, Broadcast Unreliable, Broadcast dual, and Unicast) we present:

- graphs of error versus time
- graphs of error versus number of packets
- graphs of error versus time for each node

We will present these graphs in section 5.2 for different number of nodes and for different scenarios, starting with the extreme case of a fully connected random network, and considering then other topologies like a ring network and a random network (connected). For each algorithm we present also some other graphs to see the behavior in function of different parameters, like stepsize, upgrade interval, number of nodes, topology and robustness to packet losses.

5.1.1 Tools

For doing statistics, it is necessary to parse the NS2 output (see section 4.1.1). At this purpose, there are some tools, all open source. Also in general it is useful to have all procedures as much automated as possible, so to do a big number of iterations and to plot graphs without the need of too much human control.

The different outputs are prepared through the main tcl script described in section 4.1.4 to have, in addition to the trace file and the nam file, also a time-error and an time-error for node file. From the time-error file, combined with the trace file, it is possible to derive for each iteration also the packets-error file, to take then the average and variance over a big number of iterations. Results are shown for each number of node and for different parameters. The tools we used for parsing are:

- OTcl scripting
- bash scripting
- awk scripting
- gnuplot plotting tool

all these tools are normally present in every linux distributions and they can be installed without problems in Mac OsX (they are almost all already installed) and also on WindowsXP and Windows Vista, through the cygwin package.
5.1. TYPES OF COMPARISON

Otcl scripting

We used the OTcl language inside NS2, as we describe in section 4.1.4. OTcl is not only the interface for NS2, but also a very powerful scripting language, suitable for many applications and for different using. Still for some scripting work our choice was bash scripting, for the big popularity and simplicity.

Bash scripting

Bash (http://www.gnu.org/software/bash/bash.html), acronym for Bourne-again shell, is the default shell on most Linux systems as well as on Mac OSX. It is very useful to automate also quite complex processes. We used a bash script for the complete automatisation in our work. The script calls NS2 the desired number of time for all algorithms for different parameters, and it calls also the other tools for parsing, doing statistics and plotting graphs.

Awk

Awk is a program that runs other programs written in the AWK (pronounce ‘auk’) programming language. AWK (see [4]) is a general purpose programming language that is designed for processing text-based data, either in files or data streams. For several aspects is similar to Perl, but simpler. A version of the AWK language is a standard feature of nearly every modern Unix-like operating system available today.

We wrote many AWK scripts for our work: to put data in a suitable way to plot, to take average values and variance from many iterations of the simulation, and to combine different outputs, e.g., time-error and time-packets to obtain packets-error. Awk was the main parsing tool we used.

Gnuplot

Gnuplot, [34], is a command line program that can generate two- and three-dimensional plots of function and data. We used this tool to plot all our graphs, exploiting the possibility of scripting that gnuplot offers. In an unix-like environment there are no free alternatives to using gnuplot. It would be possible to plot data in Matlab, but we preferred to use this free plot-specialized tool, that offers a lot of possibilities, even if it is not always easy to configure and to use.
Other tools

In our scripts we used also other open-source tools, mostly coming from the unix world and default installed in all unix-like environment and also on cygwin, like, e.g., the sort program to order our data, or bc to calculate automatically some parameters into the bash scripts.

5.2 Algorithms

In this section we present our result graphs for all algorithms. In figure 5.1, a and b, we see an example respectively of a ring scenario and of a random scenario we used for our simulations. To obtain a connected topology, we set properly the radio range. For all connected simulations, we set the radio range to a value bigger than the size of the topology, to be sure that all nodes are in the range of each other. For the ring topology, each node can establish communications only with two nodes (one right, one left). In the random scenario, it is all random (position of the node and radio range): was only checked the strong connection constraint. All graphs are product of the average of 100 monte carlo simulations (when there is parameters changing, 100 iterations for each value of the parameter taken in consideration). We present also the standard deviation in the time/error graphs. For naming of the algorithms, see also section 3.3.

5.2.1 Parameter setting and tuning

The performances of the different algorithms depend from several parameters. The broadcast algorithm is the one that does need less tuning, we only need to set the time-out window length to wait for acknowledgments after sending values. In our simulations, it seemed reasonable to let this parameter depend on the number of neighbors of each node, $1 + \frac{N_i}{20}$, where $N_i$ is the number of neighbors of node $i$.

For the broadcast unreliable algorithm, the most important parameter to tune is the time-out window length between each upgrade, and the number of retransmissions of the same values during this window, to have bigger probability of reception. In the simulations, the time-out window was set to 0.4 seconds, and the number of retransmissions to 3.

Also for the dual algorithm are very important the same parameters used for unreliable broadcast, length of the time-out window between each upgrade and number of retransmissions. In addition, here there is also the length of the stepsize $\mu$. In the simulations (except when differently specified) the time-out
Figure 5.1: Examples of different topologies. The radio range change in the simulations for random topology. In the ring topology each node can communicate with two neighbors (left and right).

window was set to 0.4 seconds, the number of retransmissions to 3, and the stepsize \( \mu \) to 0.02.

In the unicast algorithm, the most important parameter to set is the stepsize \( \alpha \). This was set to 0.01 in all simulations, if not differently specified.

### 5.2.2 General comparison

We start giving some preliminary comparison between the algorithms. The behavior of the different algorithms appears to be very different and not so easy to compare: all algorithms converge to a value close to the optimum, but the precision varies between the different algorithms. Also the time to obtain some level of precision varies, and the behavior changing parameters and topologies.

To compare the algorithms, we consider the absolute error referred to time and packets. We define the error at step \( k \) as \( E^k = \sum_{i=1}^{N} |\theta_i^k - \theta| \), where \( N \) is the total number of nodes, \( \theta_i^k \) is the actual value at node \( i \) at step \( k \), and \( \theta \) is the optimum value. In this section we assume a value of 0.4 as upgrading interval for Dual and Unreliable broadcast algorithm, a value of 0.01 as stepsize \( \alpha \) for Unicast algorithm and a value of 0.01 for \( \mu \) in the dual algorithm. As we see in the respective sections [5.2.3, 5.2.4, 5.2.5, 5.2.6], these are the parameters who seem to work better.

Let us present in table 5.2 a first comparison of all algorithms in a 15
node completely connected random scenario like the one in figure 5.1(b) with a radio range bigger of the diagonal size of the topology (in this case a radio range of 150 m with a topology of 100 m² in area) to be sure to have the complete connection (all node can communicate with all other nodes). We show the error value and the number of packet sent after 20 seconds.

Table 5.2: Error and packets (average over 100 simulations) after 20 seconds (completely connected scenario). Error defined as $E^k = \sum_{i=1}^{N} |\theta_i^k - \tilde{\theta}|$, step k, node i

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>TIME</th>
<th>PACKETS</th>
<th>ERROR</th>
<th>ERROR NORMALIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>20</td>
<td>6334.41</td>
<td>1.24694</td>
<td>0.08312933</td>
</tr>
<tr>
<td>Broadcast Unr</td>
<td>20</td>
<td>4855.13</td>
<td>0.2361977</td>
<td>0.015746513</td>
</tr>
<tr>
<td>Dual</td>
<td>20</td>
<td>4869.31</td>
<td>0.448073</td>
<td>0.029871533</td>
</tr>
<tr>
<td>Unicast</td>
<td>20</td>
<td>12999.9</td>
<td>1.68033</td>
<td>0.112022</td>
</tr>
</tbody>
</table>

If we analyze table 5.2, we immediately note two things:

- In the same amount of time, the unicast algorithm sends a significantly larger number of packets with respect to the other algorithms. This is because in the broadcast algorithm we do not have any waiting time, and each node sends immediately the packet after the upgrade step.

- As expected, the number of packets sent in the same amount of time in the Broadcast Unreliable algorithm and in the Dual Algorithm is similar. However in the reliable broadcast there are also the acknowledgment packets, and the number of packets sent is larger.

We note also that the error after 20 seconds for Broadcast and Unicast algorithms is larger respect to the others, but here we have different explanations for this evidence. Before explaining, let us analyze the table 5.3. Compared to the previous table, 5.2, we see that for all algorithms the number of packets sent is, as expected, a lot larger, but the error tells us that the three algorithms Broadcast Unreliable, Dual, and Unicast already had a convergence at 20 seconds, while the broadcast algorithm is improving the precision a lot, getting closer to the optimum value. So, for how it is implemented, the Broadcast algorithm is slower than the others, because it has to wait for all acknowledgments, but its precision is a lot better than the others.

About the big error of the unicast algorithm, we have to say that in this algorithm, unlike the others, we have in each node a value fluctuating around the optimum, but different at the same time for each node, so the error seems bigger than it is in reality, because of the way itself we defined the error,
\[ E^k = \sum_{i=1}^{N} |\theta^k_i - \tilde{\theta}| \] (for comparing the behavior with different number of nodes, we used the normalized error \( E^k_{\text{N}} \), with \( N \) total number of nodes, to plot the graphs). We present in table 5.4 a comparison based on the minimum error, obtained taking the minimum error at each node, no matter at what time or for which number of packets, and summing. This is only a lower bound, because we are not sure that the minimum value will be reached at the end of the simulation, maybe it is only a transient value, however we see that the unicast algorithm is comparable to the others for precision (with the exception of the reliable broadcast, which is a lot more precise).

Table 5.3: Error and packets after 50 seconds (completely connected scenario with 15 nodes). Error defined as \( E^k = \sum_{i=1}^{N} |\theta^k_i - \tilde{\theta}| \), step \( k \), node \( i \)

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>TIME</th>
<th>PACKETS</th>
<th>ERROR</th>
<th>ERROR NORMALIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>50</td>
<td>10449.1</td>
<td>0.0995448</td>
<td>0.00663632</td>
</tr>
<tr>
<td>Broadcast Unr</td>
<td>50</td>
<td>11440.7</td>
<td>0.2361995</td>
<td>0.015746</td>
</tr>
<tr>
<td>Dual</td>
<td>50</td>
<td>11397.8</td>
<td>0.447761</td>
<td>0.0298507</td>
</tr>
<tr>
<td>Unicast</td>
<td>50</td>
<td>32908.3</td>
<td>1.70994</td>
<td>0.113996</td>
</tr>
</tbody>
</table>

Table 5.4: Minimum error (completely connected scenario with 15 nodes). Min error defined as \( E_{\text{min}} = \sum_{i=1}^{N} |\min_k \theta_i - \tilde{\theta}| \), \( k \) step, node \( i \)

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MIN ERROR</th>
<th>ERROR NORMALIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>1.71405e-08</td>
<td>0.1e-9</td>
</tr>
<tr>
<td>Broadcast Unr</td>
<td>0.00521683</td>
<td>0.000347788</td>
</tr>
<tr>
<td>Dual</td>
<td>0.142558</td>
<td>0.0095038</td>
</tr>
<tr>
<td>Unicast</td>
<td>0.0709973</td>
<td>0.004733153</td>
</tr>
</tbody>
</table>

It is very useful to do a comparison also at the same number of packets: we show in table 5.5 values after 5000 and 20000 packets sent. We can see that unicast and broadcast unreliable have already reached convergence after 5000 packets, while broadcast and dual not yet. If, we look at values after 20000 packets, we see as expected that the broadcast algorithm has improved a lot the precision, the dual is improved and the others have more or less the same values. We have to notice also the difference in timing for the same number of packets. We note that the broadcast, despite the presence of the acknowledgments, sent less packets in the same amount of time respect to the other algorithms.

Graphically, in figure 5.6 is showed the same data of the previous tables. We can note the strange time packets behaviour (fig. 5.6(c)) of the reliable
CHAPTER 5. RESULTS AND COMMENTS

Table 5.5: Error and packets after 5000 and 20000 packets sent (completely connected scenario with 15 nodes). Error defined as $E^k = \sum_{i=1}^{N} |\theta_i^k - \bar{\theta}|$, step k, node i

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>TIME</th>
<th>PACKETS</th>
<th>ERROR</th>
<th>ERROR NORMALIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>13.3</td>
<td>4991.58</td>
<td>3.14384</td>
<td>0.20956</td>
</tr>
<tr>
<td>Broadcast Unr</td>
<td>20.7</td>
<td>5007.17</td>
<td>0.236198</td>
<td>0.01574652</td>
</tr>
<tr>
<td>Dual</td>
<td>20.5</td>
<td>4990.67</td>
<td>2.36547</td>
<td>0.157698</td>
</tr>
<tr>
<td>Unicast</td>
<td>7.7</td>
<td>5000.65</td>
<td>1.59591</td>
<td>0.106394</td>
</tr>
<tr>
<td>Broadcast</td>
<td>120.7</td>
<td>19998.3</td>
<td>0.00143186</td>
<td>0.000095457</td>
</tr>
<tr>
<td>Broadcast Unr</td>
<td>89</td>
<td>19989.5</td>
<td>0.2361994</td>
<td>0.15746626</td>
</tr>
<tr>
<td>Dual</td>
<td>89.1</td>
<td>19986.8</td>
<td>0.447762</td>
<td>0.0298508</td>
</tr>
<tr>
<td>Unicast</td>
<td>30.5</td>
<td>19979.3</td>
<td>1.73451</td>
<td>0.115634</td>
</tr>
</tbody>
</table>

broadcast algorithm, which diminish the inclination of the time-packets curve. We note also that the fastest convergence is from the unreliable broadcast algorithm. As we told above, the unicast algorithm seems so much worse than the others for precision, but in reality it is comparable, as it is showed in table 5.4. In figure 5.7 we see the same plots for the 5 nodes scenario, completely connected. We see that, also with a minor number of nodes, things does not change a lot, we have the fastest convergence from the broadcast unreliable algorithm, but we can see how much the reliable broadcast suffers by the bigger number of nodes in the 15 scenario completely connected. It is clear that it would be impossible to apply the reliable broadcast algorithm in a scenario in which every node has a big number of neighbors. The reliable broadcast would be too slow. The dual algorithm suffers more than the others in terms of precision and speed of convergence. In the table 5.8 we see the minimum error in the 5 nodes completely connected scenario and we notice the better precision reached by all algorithms. Comparing fig. 5.6(c) and 5.7(c) we also see that the number of packets sent by the unicast algorithm does not depend by the number of nodes. In fact, the unicast algorithm just sends packets as fast as possible, with only one node that does the upgrade step at every time.

We show the same graphs for different scenarios and for different number of nodes in figures from 5.9 to 5.12. We can see that the algorithm that seems suffer more the changes in topology seems to be the dual one. In particular in the ring topology, like in figure 5.1(a), seems to be the most difficult one, also with a smaller number of nodes. However in general all algorithms seems to have a bigger delay in convergence with the ring topology. Also the precision suffers, not only in the dual case but also with unreliable
5.2. ALGORITHMS

(a) Time error

(b) Packet error

(c) Time packets

Figure 5.6: Comparison of the algorithms in a 15 node completely connected scenario

broadcast and unicast. The best configuration seems to be the one with all connected nodes, so we could think to a network with small clusters, with node all connected inside each cluster.

Here we begin to describe results specifically for each algorithm.

5.2.3 Broadcast Algorithm

The broadcast algorithm has the value of convergence closest to the optimum value respect to the other algorithms in all scenarios. We expected this result, because it is the only algorithm between the ones we analyzed which is synchronized. There is not an explicit synchronization, but for doing the upgrade step, each node has to wait to receive new values from all its neighbor and also to receive all acknowledgments for its last sent value. So we are sure that
CHAPTER 5. RESULTS AND COMMENTS

(a) Time error

(b) Packet error

(c) Time packets

Figure 5.7: Comparison of the algorithms in a 5 node completely connected scenario

Table 5.8: Minimum error \((completely \ connected \ scenario \ with \ 5 \ nodes)\). Min error defined as \(E_{\text{min}} = \sum_{i=1}^{N} \left| \min_{k} \theta_i - \bar{\theta} \right|\), k step, node i

<table>
<thead>
<tr>
<th>ALGORITHM</th>
<th>MIN ERROR</th>
<th>ERROR NORMALIZED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broadcast</td>
<td>2.58076e-07</td>
<td>5.16151e-08</td>
</tr>
<tr>
<td>Broadcast Unr</td>
<td>0.000370062</td>
<td>7.40123e-05</td>
</tr>
<tr>
<td>Dual</td>
<td>0.00365258</td>
<td>0.000730517</td>
</tr>
<tr>
<td>Unicast</td>
<td>0.0166802</td>
<td>0.00333604</td>
</tr>
</tbody>
</table>
5.2. ALGORITHMS

Figure 5.9: Comparison of the algorithms in a 15 node Ring scenario

Figure 5.10: Comparison of the algorithms in a 5 node Ring scenario
CHAPTER 5. RESULTS AND COMMENTS

(a) Time error  
(b) Packet error

Figure 5.11: Comparison of the algorithms in a 15 node Random scenario

(a) Time error  
(b) Packet error

Figure 5.12: Comparison of the algorithms in a 5 node Random scenario
when a node does the upgrade step, it is using the correct values, so the final precision is extremely high, as we see in tables from 5.2 to 5.8 and is always improving with the time, limited only by the numerical precision we used in our simulations (10^{-6}). The problem of the broadcast algorithm seems to be the speed of convergence, not always fast respect to the others (see section 5.2.2).

In figure 5.13, we show the behavior of broadcast algorithm varying the number of nodes. We present in the graphs also the standard deviation over 1000 iterations on a completely connected random topology. We see that the number of packets sent increasing the number of total nodes in scenario is not linear (see fig. 5.13(a)), probably because of the need of more retransmissions with a bigger number of nodes. In 5.13(b), we present the average delay for iteration, where with the word \textit{iteration} we mean the delay between the first transmission of a new packet, and the arrive of the last acknowledgment from the neighbors. This type of statistic is possible only with the reliable broadcast algorithm, because in the other cases we do not have iterations with the meaning we have just explained.

While considering the very good results of this algorithm, we have to remember that it is specially tailored for the average consensus problem, and it is not general like dual optimization and unicast.

\textbf{Reliable broadcast: All connected Scenario}

As it is shown in figures from 5.6 to 5.12, the behavior of the broadcast algorithm (in red in the figures cited) is affected by topology, not so much for final precision of the value, but mainly for the convergence delay. In the completely connected scenario, we have the best behavior for this algorithm in terms of convergence delay. We show in figure 5.14 the behavior of each node in this scenario. We maintained a long time scale in the graph for comparison with other topologies and algorithms.

\textbf{Reliable broadcast: Random Scenario}

Maybe the random scenario is the most ‘realistic’ one, because we have a random topology which has the only constraint of connection, so for every topology generated, we can not know how many neighbors each node will have. If we look at the behavior for nodes in picture 5.15, we can confirm that with a random scenario the convergence delay is bigger.
CHAPTER 5. RESULTS AND COMMENTS

(a) Packets sent in function of number of nodes
(b) Average delay for iteration in function of number of nodes
(c) Convergence delay in function of number of nodes

Figure 5.13: RELIABLE BROADCAST: behavior varying the number of nodes

Reliable broadcast: Ring Scenario

The ring scenario seems the most difficult one for every algorithm. We show graphs of error for node with this topology in figure 5.16. For the reliable broadcast algorithm, differently from the others, this is not the worst topology, even if the convergence delay seems longer respect to the completely connected case.

5.2.4 Broadcast Unreliable Algorithm

The broadcast unreliable algorithm, has the fastest convergence in all situations. This algorithm, as said in the theory section, and in implemen-
5.2. ALGORITHMS

Figure 5.14: RELIABLE BROADCAST: behavior for each node in a completely connected scenario

Figure 5.15: RELIABLE BROADCAST: behavior for each node in a random topology scenario

tation section, 4.2.2, is directly derived from the reliable broadcast one, and its behavior is not completely theoretically explored. However, the results are surprisingly good. We have to remember that, as a derived of the reliable broadcast algorithm, also the unreliable case is tailored on the average consensus problem, while the dual and unicast algorithms are more general. In figure 5.17 it is shown the behavior varying waiting interval parameter. It is possible to note an improvement in precision increasing the value of waiting time, but after a value of about 0.1, it not seems possible to note further
improvements. Anyway, with bigger values of waiting time, the algorithm is more ‘robust’ in case of channel problem. It would be important in this case also to set properly the number of retransmissions during each waiting interval (set to 3 in our simulations). It is also important to note how the behavior of the algorithm seems stable also with very small (0.0005 seconds) waiting interval time. This is not true for example in the dual algorithm case.

Figure 5.16: RELIABLE BROADCAST: behavior for each node in a ring topology scenario

Figure 5.17: UNRELIABLE BROADCAST: behavior varying waiting interval parameter
5.2. ALGORITHMS

Unreliable broadcast: All connected Scenario

In the all connected scenario, we have a very fast convergence, as we show in figure 5.18. As for the other algorithms, this is the most favorable scenario. The final bias is also very small.

![Graph showing convergence](image)

(a) 15 nodes  
(b) 5 nodes

Figure 5.18: UNRELIABLE BROADCAST: behavior for each node in a completely connected scenario

Unreliable broadcast: Random Scenario

Respect to all connected scenario, here we see (figure 5.19) that the convergence delay is increased. Also the precision suffers more. It is strange in particular the differences in convergences between different nodes. It is possible to note in figure 5.19 the strange behavior of node 2. This confirm the sometimes unpredictable performances of unreliable broadcast, that is just an experiment derived from reliable broadcast.

Unreliable broadcast: Ring Scenario

For the ring scenario in unreliable broadcast (figure 5.20), we see that the rate is similar to the random case, with more uniformity between the different nodes. However, even if small, the convergence time is longer than in the all connected case.
5.2.5 Dual Optimization Algorithm

The dual algorithm has the worst behavior for convergence delay and for precision. The problem with this algorithm is to find a trade-off between precision and convergence delay, because the algorithm is very dependent on the parameters, mainly the upgrade interval window and the $\mu$ (stepsize). We show here graphs varying these two parameters. We have to say that, even if the performance of this algorithm compared to the others are not so good, this approach is very general and permits to compute different objective functions.
In Figure 5.21 it is showed the behavior of the algorithm varying the stepsize \( \mu \) and the waiting interval. Increasing the stepsize, the convergence delay is shorter but the bias is higher. Instead diminishing the waiting interval, the convergence is faster but the bias is higher, because at each upgrade step the algorithm has less new packets from other nodes. With a waiting time of 0.01 seconds the algorithm seems to be not stable (an example trajectory with waiting time 0.01 is showed in figure 5.21(c)). From a time point of view, if we decrease the waiting time interval, the amount of packets sent in the same time is of course bigger, so the convergence is faster.

(a) Packet Error (logscale) varying \( \mu \) step-size (fixed waiting interval = 0.4 seconds)  
(b) Packets Error (logscale) varying waiting interval time from 0.01 to 0.08 (fixed \( \mu = 0.02 \))  
(c) Example of trajectory of one node with waiting interval set to 0.01 (fixed \( \mu = 0.02 \))

Figure 5.21: DUAL: algorithm’s behavior varying parameters (15 nodes Completely connected)
Dual: All connected Scenario

In figure 5.22 it is showed the error for node of the dual algorithm with stepsize $\mu = 0.01$ in an all connected scenario. The dual algorithm is in general the slowest one between the algorithms proposed, but its behavior is not so bad, considering also that the algorithm can be easily adapted to other objective functions. Comparing figures 5.22, 5.23, 5.24 it is possible to see how much the dual algorithm depends on topology.

![Figure 5.22: DUAL: behavior for each node in a completely connected scenario (waiting interval = 0.4 seconds, $\mu = 0.01$)](image)

(a) 15 nodes

(b) 5 nodes

Dual: Random Scenario

The error for node results for the random scenario is showed in figure 5.23. Respect to the completely connected scenario, the convergence delay is a lot longer respect to completely connected topology.

Dual: Ring Scenario

In the ring scenario, also for the dual algorithm, we have the worst performance. We present the error for node graphs in figure 5.24. The convergence delay is the longest one.

5.2.6 Unicast Algorithm

The unicast algorithm has a behavior surprisingly good in all scenarios. We used a fixed step size instead of a diminishing one (see section 3.1), so we do not
5.2. ALGORITHMS

Figure 5.23: DUAL: behavior for each node in a random scenario (waiting interval = 0.4 seconds, $\mu = 0.01$)

Figure 5.24: DUAL: behavior for each node in a ring scenario (waiting interval = 0.4 seconds, $\mu = 0.01$)

have a perfect convergence but an oscillatory bound. The parameter to tune is $\alpha$, the stepsize. In figure 5.25 it is showed the behavior of the algorithm in a completely connected scenario varying the stepsize $\alpha$. Diminishing the value of $\alpha$, the error decreases and the convergence delay increases, even if not so much. With $\alpha < 0.01$, not reported here, there are no significant improvement in precision, but the oscillations diminish. With less nodes (figure 5.25(b)), the convergence delay does not improve a lot, but the precision seems better. The error seems bigger respect to the other algorithms because of oscillatory
behavior, but looking at the single values, the precision is comparable to dual and unreliable broadcast. Maybe the performance of the algorithm could be improved by low-pass filtering the values at each node, taking, e.g., the average over several values.

![Figure 5.25: UNICAST: behavior at changing of parameter $\alpha$](image)

**Unicast: All connected Scenario**

The behavior of the unicast algorithm with different topologies does not change a lot, however in figure 5.26 it is shown the behavior of each node in a completely connected scenario. It is possible to note the very fast convergence, and the differences in precision between different nodes.

**Unicast: Random Scenario**

In figure 5.27 it is shown the error for node of unicast algorithm in a random scenario. Differently from the other algorithms, we do not note a significant change in performance for this algorithm when the topology changes.

**Unicast: Ring Scenario**

In figure 5.28 it is shown the error for node of unicast algorithm in a ring scenario. Also here, respect to the other algorithms, it is not possible to note significant differences in performances even for the ring topology.
5.2. ALGORITHMS

Figure 5.26: UNICAST: behavior for each node in a completely connected scenario

Figure 5.27: UNICAST: behavior for each node in a random scenario
Figure 5.28: UNICAST: behavior for each node in a ring scenario
Chapter 6

Conclusions
The aim of this work was to compare different distributed optimization algorithms in a sensor network compatible with the standard IEEE 802.15.4. We developed a new application for the network simulator ns2, implementing four distributed optimization algorithms (called, following the notation used in the whole thesis, broadcast, unreliable broadcast, dual, unicast). Two of the algorithms, broadcast and unreliable broadcast are tailored upon the problem of calculating the average of a value over a sensor network; the other two, dual and unicast, are more general, and allow the optimization not only of the average, but also of other objectives function, beyond the scope in this work. Several simulations was done to test in different situations and with different parameters the behavior of the algorithms, and to compare them in different way: convergence delay in function of packets sent and time, and bias in convergence value respect to the optimum.

The results obtained do not indicate an algorithm, among the ones presented, which is the best in all situations, but instead our data show that all the algorithms have a quite good behavior in general, and that, for different application, is preferable to chose an algorithm instead of another. For example, for an application which needs high precision in calculating the average among scalar values in a sensor network (e.g., temperature), the best algorithm would be the reliable broadcast, which, thanks to the synchronization between nodes, has no final bias in convergence value respect to the optimum. If instead the problem is the speed of convergence, also for a battery saving purpose, the unicast algorithm fits better. For the unreliable broadcast algorithm, we have to say that we derived it from the reliable broadcast one, to try to avoid acknowledgments, but theoretical proof of convergence is not yet done. We included this algorithm because its behavior is surprisingly good, because the convergence is fast and enough accurate for most applications which need to calculate averages.

In general we can say that if there are too many nodes, the performances of all algorithms (maybe with the exception of the unicast one) suffer, in particular in same extreme topology as the ring one. The best results are reached in a small all-connected network, so we could think to a network split in clusters, where each cluster is composed of a small number of nodes, all-connected one with another.

Another important thing that emerges from results is that who implements a sensor network using a distributed optimization algorithm like the ones proposed in this work, in particular with the dual one, should spend some time to tune the parameters, because the performances could improve a lot tailoring in base of topology and application request parameters such as stepsize, waiting interval window, number of retransmissions.
However we think our main result was to show in a way close to reality that could be useful to utilize distributed optimization algorithms on sensor networks, and that try these algorithms on a real sensor network could bring very good results.

With the results of this thesis was written the paper by B. Johansson, C.M. Carretti, M. Johansson, [16], sent for publication (waiting for accept) at *IEEE secon 2008* conference, scheduled from 16 to 20 of June 2008 in San Francisco, USA.
Chapter 7

Future work
In this thesis we did a simulation and comparison of distributed optimization algorithms in sensor networks. It would be very good to try to implement this work on real sensors. Also we would like to investigate more deeply the problem of energy consumption in real sensors.

It would be very interesting to expand the research on other different algorithms, comparing them with the ones implemented in this thesis. To improve even more the work, we are planning to try to change objective function from least square, to see whether the results are good as well with different convex functions. It would be interesting also to try to use and to investigate more about robust optimization.

In the results of this thesis, we have found some problems in the speed of convergence if there are too many nodes. This could suggest that the best solution is to divide a sensor network into clusters, all interconnected. We would like to go deeply in this issue, to find the best way to automatically divide the network into clusters, and the best way to interconnect clusters.
Bibliography


[2] Ns by example, november 2007. URL \url{http://nile.wpi.edu/NS/}


[10] contiki. contiki os website. URL \url{http://www.sics.se/contiki/}


[28] Iyappan Ramachandran. Packets flow in ns2. It is a description who helped us to give some more details on ns2 packet’s flow. It was not possible to find again the source.


This thesis was written using \LaTeX{} on Mac OS X.

**Acknowledgments**

There are a lot of websites on internet, in which a lot of people suggest to *not* put any dedication and acknowledgments (a word so used with a different meaning during this work!) on a master thesis. I really think they are wrong, at least in my case. My experience in doing this thesis abroad was so good thanks to a lot of people that I cannot forget here. My parents was always so close to me through *skype*, with our very useful discussions every day. Also, my girlfriend Claudia was always so present for me, and it was wonderful to remain so close despite our 'double erasmus' situation, and it *is* wonderful!

A special thanks also to people who followed my work in Stockholm, in particular to prof. Mikael Johansson and to Björn, who helped me in doing this thesis. I want to remember here also all people who stayed with me in the department and in general during my experience abroad: my friends-of-a-life Lapo and Marco, who shared this and many others experiences with me, Pan,
Pigi, then Pablo, who suggested us to come in Stockholm, and all people that I have met in this period (the list would be too long!). A special thanks to my corridor neighbors, in particular Sebastien, Violaine, and Heidi. Even if probably you will never read this paper, I have to say that it was great to spend this time with you and I really hope to see you again!

Thanks also to all people who came to visit e in Stockholm, in particular, again, Claudia, my brother Pier Paolo and my cousin Alberto. It was really nice to have you with me in Sweden!

I have to say thank you also to the Guitar: it (I would like to say she) is always with me and is a very important part of my life, and gives me a lot of ideas also in engineering!

La laurea è anche la fine di un ciclo, e, senza discorsi troppo lunghi e troppo retorici, vorrei ringraziare Feriae Matricularum per tutte le esperienze irripetibili e uniche che mi ha fatto vivere, e per tutte le esperienze extra-universitarie che tanto mi hanno insegnato. La favola non finisce qui...Grazie a tutti.