Evaluation of the Received Signal Strength Indicator for Node Localization in Wireless Sensor Networks

Mémoire présenté
à la Faculté des études supérieures de l’Université Laval
dans le cadre du programme de maîtrise en informatique
pour l’obtention du grade de Maître ès Sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUEBEC

2009

©Siarhei Smolau, 2009
Abstract

A wireless sensor network (WSN) consists of a large number of sensor nodes that are capable of detecting many types of information from the environment, including temperature, light, humidity, radiation and seismic vibrations. Current applications of WSNs include: physical security, air traffic control, video surveillance, environment and building monitoring. Such applications require that each sensor node knows its exact location. In this context, the received signal strength indicator (RSSI) is often used for distance measurements between the sensor nodes. This thesis presents a method for the evaluation of the RSSI properties in application to node localization in WSN. More specifically, a WSN application is implemented for collecting RSSI measurement in different conditions. The application consists of two parts: an experiment control script which runs on a computer, and an experiment mote firmware which runs on each WSN node. Statistical analysis of variance (ANOVA) was performed to determine the factors affecting the RSSI measurements. Result analysis shows that: the relation between RSSI values and distances depends on the environment; the used WSN motes are manufactured with enough precision, as the differences between the motes are insignificant; even if the RSSI measurements have significant variation, the mean RSSI values correlate with the distances; using different transmission power levels can provide additional information about the distances.
To my parents, Zoja Konstantinovna and Vasily Nikiforovich Smolau.
I would like to thank my advisor, Dr. Ronald Beaubrun, for his help, guidance and patience.

I would like to thank my family, Zoja and Aliaksandr Smolau, Ludmila and Sergey Brichkovskij, Pierre and Olivette Champagne for their support and encouragement. Special thank to my beloved Julie for her love and for her expertise in mathematical statistics.

I would like to thank Dr. Adam Schultz and Philip Taylor from Earth-Ocean Systems, who influenced my choice of the subject.

I would like to thank Dr. Robert Rousseau and Dr. Sebastien Tremblay, people from Research Group in Cognitive Psychology, School of psychology, University Laval, for inspiration.
# Contents

Abstract ii

Acknowledgments iv

Contents v

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Basic concepts 1

1.2 Problematic 3

1.3 Research objectives 4

1.4 Organization 4

2 Characterization of WSN applications 5

2.1 WSN usage 5

2.2 Event detection 9

2.3 Periodic measurements 11

2.4 Function approximation 15

2.5 Tracking 18

3 Node localization in wireless sensor networks 22

3.1 Ranging methods 22

3.1.1 Time of Arrival (ToA) 23

3.1.2 Time Difference of Arrival (TDoA) 23

3.1.3 Received Signal Strength Indicator (RSSI) 24

3.2 Geometric positioning methods 25

3.2.1 Lateration 25

3.2.2 Angulation 28

3.2.3 Multi-Dimensional Scaling 29

3.3 Probabilistic positioning methods 31
3.3.1 Location fingerprinting ....................................... 31
3.3.2 Monte Carlo localization .................................... 31
3.4 Existing localization algorithms ............................... 33
  3.4.1 Categorization ............................................... 33
  3.4.2 Ad hoc Positioning System ................................. 35
  3.4.3 Anchor-Free Localization ................................. 36
  3.4.4 MoteTrack ................................................. 37

4 Implementation procedures, experiments and results 39
  4.1 Experiment design ............................................. 39
    4.1.1 Goals and hypotheses .................................. 40
    4.1.2 Preliminary concepts .................................... 40
  4.2 Implementation of the WSN application ..................... 41
    4.2.1 Hardware and software platform ....................... 41
    4.2.2 Requirements and deployment .......................... 42
    4.2.3 Application behavior ................................... 43
    4.2.4 Architecture and implementation of the experiment control script 50
    4.2.5 Architecture and implementation of the experiment mote firmware 51
    4.2.6 Communication protocol ................................ 51
  4.3 Experiment results ............................................ 52
    4.3.1 Experiment 1: Transmission ranges ...................... 52
    4.3.2 Experiment 2: Variability in motes behavior .......... 59
    4.3.3 Experiment 3: Impact of the environment ............... 68

5 Conclusion 78
  5.1 Summary ....................................................... 78
  5.2 Limitations ................................................... 80
  5.3 Future work .................................................. 80

Bibliography 82

A Experiment mote firmware source code 91

B Collected experiment measurements 123

Index 126
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Different roles of sensor nodes.</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Ranging using time of arrival (ToA) method</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Ranging using time difference of arrival (TDoA) method</td>
<td>24</td>
</tr>
<tr>
<td>3.3</td>
<td>Using trilateration to determine the position of a node</td>
<td>26</td>
</tr>
<tr>
<td>3.4</td>
<td>Using multilateration for position evaluation based on imprecise ranging</td>
<td>28</td>
</tr>
<tr>
<td>3.5</td>
<td>Using triangulation to determine a node position</td>
<td>29</td>
</tr>
<tr>
<td>3.6</td>
<td>Graph rigidity: 1 - non-rigid graph; 2 - rigid but not globally rigid graph;</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>3 - globally rigid graph</td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Deployment diagram</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Activity diagram</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>State machine diagram for the experiment control script software behavior</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>State machine diagram for the experiment mote firmware behavior</td>
<td>49</td>
</tr>
<tr>
<td>4.5</td>
<td>Boxplot representation of the probability density function (pdf) of a normal</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>population</td>
<td></td>
</tr>
<tr>
<td>4.6</td>
<td>RSSI vs Distance, for TX=3,4,5</td>
<td>55</td>
</tr>
<tr>
<td>4.7</td>
<td>RSSI vs Distance, TX=9,10,11</td>
<td>56</td>
</tr>
<tr>
<td>4.8</td>
<td>RSSI vs Distance, TX=19,20,21</td>
<td>57</td>
</tr>
<tr>
<td>4.9</td>
<td>RSSI vs Distance, TX=29,30,31</td>
<td>58</td>
</tr>
<tr>
<td>4.10</td>
<td>RSSI vs Distance for all motes for TX=3</td>
<td>60</td>
</tr>
<tr>
<td>4.11</td>
<td>RSSI vs Distance for all motes for TX=5</td>
<td>61</td>
</tr>
<tr>
<td>4.12</td>
<td>RSSI vs Distance for all motes for TX=10</td>
<td>62</td>
</tr>
<tr>
<td>4.13</td>
<td>RSSI vs Distance for all motes for TX=20</td>
<td>63</td>
</tr>
<tr>
<td>4.14</td>
<td>RSSI vs Distance for all motes for TX=30</td>
<td>64</td>
</tr>
<tr>
<td>4.15</td>
<td>RSSI vs Distance for 2 environments with TX=5</td>
<td>69</td>
</tr>
<tr>
<td>4.16</td>
<td>RSSI vs Distance for 2 environments with TX=10</td>
<td>70</td>
</tr>
<tr>
<td>4.17</td>
<td>RSSI vs Distance for 2 environments with TX=20</td>
<td>71</td>
</tr>
<tr>
<td>4.18</td>
<td>RSSI vs Distance for 2 environments with TX=30</td>
<td>72</td>
</tr>
<tr>
<td>A.1</td>
<td>Main experiment mote firmware implementation wiring diagram</td>
<td>92</td>
</tr>
<tr>
<td>A.2</td>
<td>Mote behavior implementation wiring diagram</td>
<td>95</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Variation of RSSI as a function of the distance. .......................... 53
4.2 Variation of RSSI due to the motes difference (reception) .......... 68
4.3 Variation of RSSI due to the motes difference (transmission) ....... 68
4.4 Variation of RSSI due to the environment (reception) ............... 75
4.5 Variation of RSSI due to the environment (transmission) .......... 75
4.6 Differences between motes (classroom, reception) ................. 76
4.7 Differences between motes (classroom, transmission) .............. 76
4.8 Differences between motes (tunnel, reception) ....................... 76
4.9 Differences between motes (tunnel, transmission) ................. 77

B.1 Experiment 1. Means and standard deviations of RSSI measurements. . 124
B.2 Experiment 2. Means and standard deviations of RSSI measurements. . 125
B.3 Experiment 3. Means and standard deviations of RSSI measurements (classroom). .................................................. 125
B.4 Experiment 4. Means and standard deviations of RSSI measurements (tunnel). .................................................. 125
Chapter 1

Introduction

Recent advances in radio technologies have enabled the proliferation of small electronic devices which are able to detect many types of information from the environment, including temperature, light, humidity, radiation, the presence or nature of biological organisms, seismic vibrations, and more. Such devices are commonly called sensor nodes (or simply sensors), and can be deployed on the ground, on bodies, in vehicles, inside buildings. They are capable of collecting, processing and communicating information with each other, as they may be wirelessly connected to form a wireless sensor network (WSN) [56]. A wireless sensor network consists of a large number of sensor nodes that may be randomly and densely deployed. Current applications of sensor networks include: military sensing, physical security, air traffic control, video surveillance, environment and building monitoring [75], [38], [58]. Such applications require that each node knows its exact location. This thesis proposes a method for node localization in a real WSN.

The rest of this chapter is organized as follows. The first section introduces basic concepts needed for understanding the thesis. The second section presents the problematic of the thesis. The third section lists research objectives, whereas the last section gives the organization of the thesis.

1.1 Basic concepts

A wireless sensor network (WSN) is a computer network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or en-
environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations [17]. Often, the information provided by the network is only useful if it is possible to determine to which part of the environment this information corresponds. For example, imagine a sensor network which monitors a forest to detect fire by measuring the air temperature. A useful outcome from this network would be not only the signal that the air temperature is too high, but also in which part of the monitored forest the fire is started, which enables to fight the fire more efficiently. Since wireless sensor networks may be deployed in inaccessible terrains or disaster relief operations, the position of sensor nodes may not be predetermined. As a result, a localization system is required in order to provide position information to the nodes. Sometimes, sensor node localization is indispensable in event observation, or object tracking.

Recently, many localization algorithms for WSN have been proposed [40], [52], [59], [60]. Among them are the range-based approaches which need to measure point-to-point distances or angles, in order to estimate the location. Such estimates constitute an important component of localization systems, because they are used by both the position computation and localization algorithm components. In this context, four methods are mainly used by localization systems for distance measurements [61], [52]: received signal strength indicator (RSSI), time of arrival (ToA), time difference of arrival (TDoA), and angle of arrival (AoA).

RSSI is one of the simplest approaches that has been used for estimation of distances between nodes. The distance estimates are based on the strength of the signal received by another node. In this context, a sender node sends a signal with a determined strength that decreases as the signal propagates. The bigger the distance to the receiver node, the less the signal strength when it arrives at that node. The main advantage of this method is its low cost, since most receivers are capable of estimating the received signal strength. However, this method is very susceptible to noise and interference, which results in high inaccuracy in distance estimation. Although this approach has been demonstrated to perform poorly, it is the most used solution for distance estimation [2], [42], [10].

Moreover, distances between the nodes can be estimated more accurately by estimating the time difference between transmission and reception of slow travelling signals, such as ultrasound or acoustic waves. By transmitting both radio and ultrasound/acoustic waves at the same time, a receiver node has a transmission reference time stamp by which to calculate the time difference. The drawback of this approach is the need for nodes to have additional hardware for dealing with acoustic signals. Whereas the previous approaches have been based on distance estimates between nodes, location can
also be derived from the angle of arrival (AoA) knowledge between nodes. Techniques for obtaining angle estimates include the use of phased arrays of radio or ultrasonic receivers, or by using rotating directional beacons [49].

1.2 Problematic

The choice of which method to use to estimate distances between the nodes in a WSN is an important factor that influences the global performance of the network. In general, to estimate a position, a node uses at least three distance estimations, each with an associated error [51]. In this case, if only the accuracy of these methods was important, we could use TDoA since it has better precision. However, the size and cost of the nodes must also be taken into account. Thus, the method chosen for estimating distances depends on the application requirements, as well as the available resources. In other words, accurate ranging methods require specialized and expensive hardware.

On the other hand, RSSI is one of the simplest and cheapest ranging methods, as RSSI information is always available in practically all transceivers suitable for wireless networks. More specifically, if the signal path loss is modeled correctly, the RSSI can successfully depict the relationship between the distance and the received signal strength. However, there are several problems. First, each time a sensor network is installed, the system needs a RSSI calibration, since the path loss model can be changed according to the environment. Second, RSSI is very unstable. It can be affected by not only the environment, but also obstacles or people. If all the nodes are calibrated once, there is no guarantee that the same parameters will fit to a changing environment. In this context, the RSSI measurements may be inadequate if the physical characteristics of the environment are not taken into account. Then, how can the physical environment be taken into account when using RSSI for distance measurements in WSN?

Theoretically, the signal strength is inversely proportional to the squared distance, and a known radio propagation model can be used to convert the signal strength into distance. However, in real-world environments, this indicator is highly influenced by noises and obstacles, which makes it hard to model mathematically. In these cases, it is common to make a system calibration, where the RSSI values and distances are evaluated in a controlled environment. But, is it really important to calibrate the sensor nodes for distance measurements in different environments?
1.3 Research objectives

The main objective of this thesis is to elaborate a set of scenarios and experiments which enable to evaluate the main factors that influence the RSSI and the distances in the context of node localization in WSN. More specifically, we aim at:

- Characterizing WSN and their applications;
- Identifying the main parameters for node localization in a WSN;
- Evaluating the impact of environment on the RSSI and the distance estimates. In this context, each node may be calibrated in function of the environment in order to produce more accurate measurements.

1.4 Organization

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the WSN applications. Chapter 3 explains the problem of localization. Chapter 4 describes the implementation procedure and results. Chapter 5 makes a conclusion based on results obtained, discusses limitations of the solution, and provides suggestions for further works.
Chapter 2

Characterization of WSN applications

Despite wireless sensor networks represent a new field, some systems were already implemented, and applications of wireless sensor networks promise many advantages over traditional solutions. This suggests that WSN will be widespread in the future, i.e. they will be used in different areas of human life. This chapter presents an overview of the WSN applications. It is structured as follows. The first section describes the usage of WSN, discusses their advantages, and categorizes their applications. The second section presents WSN applications implementing event detection. The third section presents WSN applications implementing periodic measurements. The fourth section presents WSN applications implementing function approximation. The fifth section presents WSN applications implementing tracking.

2.1 WSN usage

The purpose of wireless sensor networks is to translate information provided by the environment into digital form, collect it and provide it to other computers [56]. Traditionally, data are gathered by doing surveys or by installing wired sensing infrastructure. In a survey, a measuring equipment is installed at the place of interest, then it collects data, then the equipment is brought back to laboratory, and the collected data are analyzed. Sophisticated scientific instruments, like telescopes, satellites, or submarines, are used for this purpose. The process of doing the survey often involves manual operation and asks for well-trained personnel. This makes this approach expensive. As a result, only limited areas of the environment can be covered. Another drawback is that the data about the region of interest are available only for the period of survey.
Chapter 2. Characterization of WSN applications

For example, Werner-Allen et al. [76] discuss the deployment of sensor networks for monitoring volcanos. As stated, “Today's typical volcanic data-collection station consists of a group of bulky, heavy, power-hungry components that are difficult to move and require car batteries for power. Remote deployments often require vehicle or helicopter assistance for equipment installation and maintenance. Local storage is also a limiting factor: stations typically log data to a “Compact Flash” card or hard drive, which researchers must periodically retrieve, requiring them to regularly return to each station.”  And “The increased scale promised by lighter, faster-to-deploy equipment will help address scientific questions beyond current equipment’s practical reach.”

Wireless sensor networks promise several advantages over traditional wired sensor networks or big monolithic sensing instruments [35]. These advantages are: easier deployment, higher reliability, lower cost, higher quality of measurements and network processing. Because of the absence of wire, the setup and installation of a wireless sensor network can be performed quicker, cheaper and in a more unobtrusive way. Also, the absence of wire allows the network setup to be more flexible. Furthermore, wireless sensor networks can be installed in a hostile and remote environments, where the installation of conventional networks is not possible. In particular, Vasilescu et al. [72] discuss using of wireless sensor networks for underwater measurements. The absence of wire and the presence of mobile nodes in form of autonomous underwater vehicles (AUV) allow relocating the wireless sensor network after deployment.

Moreover, dense deployment of many nodes covering the same location creates redundancy which provides greater fault-tolerance [3]. If a small portion of the sensors fails, the network can continue to provide the data, assuming that defective sensors can be excluded from the communication, and the sensors are calibrated, either individually or collectively, either before deployment or continuously in their environment [73]. Articles [54], [68], [45] describe the deployment of wireless sensor networks for habitat monitoring at Great Duck Island and show the implementation of the network health-monitoring system which allows to filter out data from the failed nodes.

The cost of deployment and operation of wireless sensor networks is lowered by eliminating the cost of wiring. Also, a great number of simple and cheap sensors are supposed to be more cost-effective than a small number of complex and expensive sensors. Take, for example, the market of predicted machinery maintenance, discussed in [37]. The goal is to gather data about the state of the machinery to determine when such machinery needs to be repaired. Currently, two methods are used: manual data collection and online surveillance. In the first case, data are collected manually using handheld instruments, like infrared thermometers or ultrasound detectors. In the second case, hard-wired sensors are utilized for data acquisition - each unit processes the
Chapter 2. Characterization of WSN applications

data and delivers it across wired network to a central repository. The article states that an industry cross section shows that wired sensor networks penetration into the market is less than 10%, primarily due to the cost. In the remaining 90% of the market, 20% use manual data collection, and most of them are not happy with the level of prediction and correlation they provide.

Small size and big quantity of installed wireless sensor network nodes allows sensors to be installed closer to studied phenomena, which provides higher spatial resolution. Absence of wire allows to cover bigger areas than it would be economically feasible with a wired sensor networks. Distributed sensing provides robustness to environmental obstructions [19].

The computational abilities of each node provide distributed data processing, making the network to gather only relevant information. Article [14] discusses the usage of wireless sensor networks for structural health monitoring. Sensors are placed in a building, bridges, off-shore oil-rigs to localize the damage. Several structural health monitoring techniques do not use the data directly in its raw-form. For example, time series based damage detection techniques use the auto-regressive/auto-regressive moving average (AR/ARMA) coefficients [14]. These can be calculated locally at each sensor node and transmitted to the central location, instead of transmitting the entire data. Thus, transmitting 40 complex ARMA coefficients instead of 5000 samples of data leads to more than 99% savings in communication overhead.

WSN nodes can measure many different physical parameters (temperature, pressure, light etc.). Different industries require different information to be sensed. This leads to the vast design space and the very different requirements for various applications. A detailed review of the WSN applications for different industries can be found in [1, 9] (underwater surveillance), [75] (agriculture), [20] (transport). The long list of applications, sorted by industry, is not very informative, because within one industry, different applications can have different requirements, and two applications from different industries can be in fact, quite similar in structure, varying only in the meaning of collected data.

There are several ways to categorize WSN applications. Romer and Mattern [58] suggest to categorize WSN applications along several dimensions: deployment, mobility, resources, heterogeneity, communication modality, infrastructure, network topology, coverage, connectivity, network size, lifetime and quality of service. The analysis of fifteen WSN applications shows that this approach is useful for discussing requirements, though it may be too complex for the purpose of simple review. Kuorilehto et al. [38] try to categorize applications based on their type, requirements, amount and frequency
of collected data, scale and density. Holger and Willig in [27] observe that many applications share similar typical scenarios that WSN nodes are following to exchange the data. Those scenarios are called *interaction patterns*. A piece of the relevant information is called *event*. The actual node that can sense data is called the *source*. A node where the data should be delivered to is called *sink*. Sometimes, these sinks are parts of the sensor network itself; sometimes they are clearly systems “outside” the network (e.g., PDA communicating with WSN or *gateway* to another network where data will be processed and stored). When a source node has limited transmission power, or it is not located along the line of sight of a sink, *multi-hop communication* is used. In the context of multi-hop communication, each node between a sink and a source is acting as a *relay*, dispatching information towards the destination. Figure 2.1 illustrates the role of each sensor node in a wireless sensor network. Four relevant interaction patterns are listed: event detection, periodic measurements, function approximation, and tracking. These interactions can change dynamically over time, or can take place only for one specific request of the sink. Implementation of different interaction patterns often requires different trade-offs and different solutions. Thus, it may be a good way to group applications. In the following sections, examples of the applications implementing each interaction pattern will be provided.

![Diagram of sensor node roles](image)

**Figure 2.1**: Different roles of sensor nodes.
2.2 Event detection

In *event detection*, sensor nodes should send data to the sink(s) as soon as they have detected the occurrence of a specified event. The simplest events can be detected locally by a single sensor node in isolation (e.g., a temperature threshold is exceeded); more complicated types of events require the collaboration of nearby or even remote sensors to decide whether a (composite) event has occurred (e.g., a temperature gradient becomes too steep). If several events occur at the same time, *event classification* might be an additional issue.

Lédeczi *et al.* [41] describe using a wireless sensor network to detect sniper’s position and trajectory of their bullets. The sensors are able to detect two kinds of events: an acoustic muzzle blast and a ballistic shockwave. Muzzle blast is a loud characteristic noise originating from the end of the muzzle, and propagating spherically from the end of the muzzle at the speed of sound. It makes it suitable for localization purposes, though it can be suppressed by silencers or masked by sound propagation effects. The shockwave is the result of the air being greatly compressed at the tip and expanded at the end of the bullet, as it slices through the air. The ideal shockwave front is a cone, moving along the trajectory of a projectile.

In [41], the sensor nodes hardware was built based on *MICA2* mote device, extended with custom-made sensor board, implementing the signal processing algorithms processing the output from low-cost microphones. The motes were controlled by TinyOS operating system, with a service implementing routing and integrated time synchronization. The data from individual sensors were sent to a central sink which was producing sensor fusion and displaying the result of localization. A network of 56 sensors was deployed for the field tests. Sensors were deployed manually with approximately 20-30 cm precision.

In the muzzle blast fusion test, the system was able to localize 83% of shots with an error of less than one meter, and 98% of shots with precision of less than two meters, the average error was 0.6 m. In the shockwave fusion test, the sensor network of 60 motes covered an area of 80 m by 80 m. In one experiment, 12 shots were fired over the middle of the network, so that there were sensors on both sides of the trajectory. The average azimuth error was 0.66 degree, the average elevation error was 0.61 degree, and the average range error was 2.56 meters. Another 11 shots were fired from the same distance near the edge of the network, so that there were no or only a few sensors on one side of the trajectory. The average error increased to 1.41 degree in azimuth, to 1.11 degree in elevation and to 6.04 meters in range. In a test with two simultaneous shots,
the system was able to localize both trajectories with an efficiency of 50%. The authors point that wireless sensor networks have clear advantages over traditional centralized acoustic-array instruments because they can mitigate acoustic multipath effects, and they can also resolve multiple simultaneous shots.

In [76], Werner-Allen et al. describe using the WSN for monitoring a volcano. A network of 16 nodes equipped with seismoacoustic sensors was deployed on Volcán Reventador in Ecuador. Each node was based on moteiv TMote Sky device equipped with a seismometer and a microphone, and powered by a pair of alkaline D-cell batteries. Analytical methods used to process signals extracted by volcanic data-collection networks require that a wireless sensor network produces highly reliable data - a single missed or corrupted sample can invalidate an entire record. Small differences in sampling rates between two nodes can also invalidate analysis, so samples need to be accurately time-stamped. Also, much of the data analysis focuses on discrete events, such as eruptions, earthquakes, or tremor activity. Many interesting signals at Reventador had a duration of less than 60 seconds and occurred several dozen times a day. Thus, the network was designed to capture time-limited events, rather than continuous signals.

Each node was sampling two or four channels of seismoacoustic data at 100 Hz, storing the samples into local flash memory. The nodes were transmitting periodic status messages and performing time synchronization. The node’s flash memory fills in roughly 20 minutes when recording two channels at 100 Hz. Taking into account overheads caused by packet framing, medium access control, and multihop routing, the achievable data rate is less than 10 Kbytes per second. Thus, each node can gather data faster than it can transmit it. The amount of data is so large, that logging data to local storage for later retrieval is not feasible. So, each node stores sampled data in its local flash memory, which is treated as a circular buffer. Each block of data is time-stamped using the local node time. The local time is then mapped into global time. Each node runs an event detection algorithm on the sampled data. When a node detects an interesting event, it sends a message to the base station. If enough nodes report an event within a short time interval, the base station initiates data collection. This filtering prevents detection of a spurious event from triggering data collection. The base station downloads 30-60 seconds of data from each node, ensuring that the system gathers all buffered data from the event. When data collection is completed, nodes return to sampling and storing sensor data. This mode of operation is a result of a compromise between the requirement of application for high-data-rate sensing and the low radio bandwidth available for each WSN node.

The command and control interface was developed to monitor the network state and
setting data collection parameters, assisting the deployment and aiding in resolution of connectivity problems. The deployment had a duration of 19 days. It was possible to retrieve data from the network 61% of the time. During the deployment, the network detected 230 eruptions and other volcanic events, and logged about 107 Mbytes of data. Authors state that using WSN permitted to create an equipment which is smaller, lighter and consumes less power. This is essential, because remote deployments often demand vehicle or helicopter assistance for installation and maintenance. Thus, weight, volume and local storage are limiting factors, as lighter and faster-to-deploy instruments promise to perform experiments of larger scale.

2.3 Periodic measurements

In periodic measurements, sensors are periodically reporting measured values. Often, these reports can be triggered by a detected event. The time period between consecutive reports is application-dependent. Several reasons explain why it is necessary to perform periodic measurements and to send data to a central server instead of in-network processing:

- to perform trend analysis, all data need to be collected at a central location;
- it is essential to enable direct comparison of measured data against the manual system;
- many algorithms which are used to predict equipment failure are implemented in a proprietary back-end software.

In [37], Krishnamurthy et al. describe using WSN for preventive equipment maintenance. Predictive maintenance (PdM) is the general term applied to a family of technologies used to monitor and assess the “health” of a piece of equipment (e.g., a motor, a compressor, or a cooler) that is in service. PdM technologies allow the user to detect the most impending failures well in advance. The chosen PdM technique is the vibration analysis which presumes that source vibration frequencies can be identified and assigned to specific components of the test subject. The analysis of time domain and frequency domain waveforms is then used to identify changes in amplitude and frequency patterns, suggesting repair or replacement. Two hardware platforms were used to implement WSN for vibration analysis in two different environments. The first environment was a central utility support building at a semiconductor fabrication plant.
The second environment was an engineering space aboard an oil tanker operating in the North Sea. The requirements for the WSN were:

- fault tolerance and reliability: data from the sensors must be accurate and delivered in a timely manner, as the network should be robust to temperature, humidity and vibrations;
- long-lived battery-powered operation, since operating and safety regulations require that each piece of equipment should have dedicated power circuit;
- maintainability: the sensor network should provide a user interface that enables the diagnostic of each sensor;
- integration into existing applications: existing PdM tools should be used for end-user interface and data analysis;
- security: as modified or falsified data from the sensor network could have dangerous effects in industrial environment, ensuring data integrity, confidentiality and authenticity is important.

The system was using Wilcoxon model 786A accelerometer sensors. Each sensor is calibrated to 100 mV/G with 5% calibration sensitivity at 25°C. The dynamic range of the accelerometers is 80g’s peak with a maximum frequency range of 30 kHz. The sensor board was developed with a sampling rate of 19.2 kHz, allowing for a frequency range of 9.6 kHz. On each sensor board, a subset of channels is connected to vibration sensors. Each sample consists of 6 kB of time-series data from the vibration sensor, for a total of 36 kB per node. The data are captured from all sensors at regular intervals and sent to a server for analysis.

The application consists of a hierarchical communication structure, a cluster-based power management protocol, and a reliable bulk transport. This allows to coordinate periodic data collection across a large number of sensing points while maximizing sleep time, i.e. the time when nodes are in stand-by mode, having their hardware switched off to conserve power. In a trial deployment, 150 accelerometers, 26 sensor nodes, 4 gateway nodes and 1 PC were used. The site survey was performed to check the environment for connectivity. The lifetime experiments were conducted. In a ship experiment, the sensors were set to be in sleep mode for 18 hours. At this rate, the network was expected to produce good data for over 82 days with a sample period of about 20.5 hours. In a plant experiment, the sleep mode duration was set to 5 hours, which was calculated to allow at least 21 days of good results with a sample period of about 7 hours. In practice, the ship experiment network was able to perform data
collection for 19 weeks, and the plant experiment was producing data for 6 weeks. The majority of nodes successfully delivered results in at least 80% of the time. Authors conclude that their trial was able to retrieve data of sufficient quantity to demonstrate that WSN are indeed useful for the PdM applications. Also, it was noted that sensor networks can provide high quality data at a relatively low investment in installation and operation.

In [12], Chintalapudi et al. describe the construction of WSN for structural health monitoring (SHM). SHM is an area of research aiming to develop systems that can autonomously and proactively assess the structural integrity of bridges, buildings and aerospace vehicles. SHM research attempts to use sensors to localize damage and detect its size by measuring temporal patterns of vibrations induced in the structure. Using wireless sensor networks in this field allows increasing spatial resolution of systems, avoiding high costs of wiring and providing high quick installation of the system.

The SHM application domain introduces specific requirements on the sensing software. In particular, many SHM applications cannot tolerate data loss, as sensor data should be transmitted with absolute reliability. Also, signal-processing techniques use temporal correlation for data analysis, thus the sensor clocks need to be synchronized. (Some algorithms demand synchronization within tens of microseconds.) SHM sensors generate data at high rates, comparable to the nominal rates of modern low-power radios. Large structures typically comprise hundreds of parts, where each part will require at least two tri-axial acceleration sensors. A single sensor node generating 16-bit vibration data along three axes at 500 samples per second can easily consume 25% of the nominal data rate of the IEEE 802.15.4 low-power radio. Thus, it is infeasible to continuously collect raw sensor data at a central computer for analysis.

Two WSN-based systems were implemented: Wisden and netSHM [12]. Wisden is a data acquisition system which delivers time-synchronized structural-response data reliably from several locations over multiple hops to a base station. It can be used in the following scenario. A transportation agency is ready to declare a newly built bridge open, and allows a team of structural engineers to measure its structural properties for one or two days. A wireless network can be deployed in tens of minutes, but it is often unknown where exactly to instrument the structure, because the structural characteristics might not be precisely known. Wisden allows the engineers to move the sensors in order to determine appropriate locations.

Wisden deployment consists of tens of wireless nodes. Nodes are self-configured to form a tree-topology and reliably send time-synchronized vibration data to the sink, potentially over multiple hops. The sink forwards the data to a base station. Sensors can
be moved or turned on/off in a working Wisden deployment. Nodes are based on Mica-2 or Mica-Z hardware motes from Crossbow Technologies which measure structural vibration using highly sensitive tri-axial accelerometers. Wisden implements a negative acknowledgment hybrid hop-by-hop/end-to-end reliability scheme. The system observes gaps in received sequence numbers and retransmits lost packets from a message cache, providing hop-to-hop reliability. This allows to overcome the high (up to 30 percents) wireless message loss rates, often observed in a real environment. End-to-end reliability is needed because hop-by-hop reliability alone cannot recover lost packages when the topology changes; so the copy of every generated packet is also stored in the source node’s flash memory for retransmission. The base station keeps track of missing packets from all nodes and notifies the appropriate sensor nodes to retransmit them as needed. The messages are time-stamped at the base station. Instead of synchronizing clocks across the network, the sink estimates message generation times according to the each source’s node local time. Finally, Wisden implements a detection-based compression scheme that lets the nodes detect a significant event’s occurrence and transmit only data corresponding to that event.

Wisden was deployed in two real environments: a seismic test structure and an abandoned office building in Los Angeles [12]. The results of deployment show that a typical structural response due to a sudden impact lasts less than a second, so SHM applications should support high data rates. The flash memory access time becomes the bottleneck that limits the sampling rate. The communication environment in the building was noticeably worse than in the seismic test structure. The overall average message reception rate was 81.12%, but for some links, it was as low as 37.6%. Frequent route changes were observed. The deployment was proven to be successful. It is noted that using WSN provides flexibility and ease of deployment: traditional data acquisition system took several days to set up, whereas Wisden required roughly 30 minutes.

netSHM is a programmable sensor-actuator system which allows to implement algorithms in a higher-level language, like Matlab and C. netSHM implements the so-called forced excitation, which means that the system has actuators - shakers or impact hammers which can provide external forces to the tested structure at a specified moment of time. This allows to schedule tests to be performed at specific times (say once a day), thus operating network at low-duty cycle, where sensors sleep most of the time, promising long-lifetime of the system. netSHM uses a two-tier hierarchical architecture: resource-constrained wireless sensor nodes (Mica-Z) form a mote-class tier, whereas more endowed nodes (embedded systems with 32-bit processors, 802.1x radios, and several megabytes of flash memory) form the upper tier. The high-end nodes is configured to provide a high-capacity backbone. In netSHM, the application-specific code is residing only in the upper-tier nodes, which issue commands to mote-class nodes via a
task interface to sample sensors at a specified rate or perform basic signal processing on
a data. As a result, the networking code becomes generic, and can be implemented as
reusable middleware. netSHM was deployed on two structures: a scaled model of a four-
floor building and a seismic test structure. The deployment was using 15 Mica-Z motes
and two Stargates (high-end nodes). The damage detection and damage localization
algorithms were implemented. Reliable delivery of data was achieved.

Researchers anticipate that networked sensing and actuation will replace wired in-
strumentation in structural testing over the next three to five years [12]. Initial deplo-
yments will be specialized and have a small scale, large-scale structure sensing demands
development of proper suitable technology, which is predicted to be achieved in five to
ten years.

2.4 Function approximation

In the case of function approximation and edge detection, a WSN is used to approximate
and extract spatial characteristics of the measurement function, using a limited number
of samples taken from each individual sensor node [27]. This approximate mapping is
made available to the sink. How and when to update this mapping depends on the
application’s needs, as do the approximation accuracy and the inherent trade-off against
energy consumption. The similar problem of edge detection is the task of finding areas
or points of the same given value. As example, for agriculture, the task may be to find
the area of the field with a high risk of a plant disease to apply a treatment [39].

In [80], the authors describe a WSN used to detect extreme temperature gradi-
ents for marine microorganism monitoring. The regions of sharp temperature change
(thermoclines) in oceans are a breeding ground for certain marine microorganisms. Mi-
croorganisms, such as phytoplankton, are exceedingly small, and are distributed in the
ocean at varying scales. It is not practical to locate them by measuring their density
everywhere. For locating microorganisms and for studying their behavior, it is needed
to study how their numbers and location are correlated with chemical (nutrient concen-
tration) and physical parameters (temperature, light intensity). There are two major
factors: light and nutrients. In the zone where there is good balance between these
two factors, the density of certain microorganisms may be expected to be high. Such
a region can be a thermocline, i.e. a zone where seawater temperature drops rapidly.
This sharp change in temperature acts as natural barrier to nutrient diffusion.

The described system implements distributed adaptive sampling algorithms based
The nodes of wireless sensor network are allowed to move to increase the density of the sensors in the area of interest. But, communication range in water is limited, as each node will be able to communicate with only its nearby neighbors. The algorithm works as follows: $n$ nodes are deployed in a vertical array where the topmost node is connected to the external node. Each node has a pressure sensor, as it is able to estimate its depth. The nodes are moving by changing their buoyancy. The search space is divided into regions and each node can move to explore one such region. Data from each node are aggregated on its way to the topmost node to combine the conclusion about the thermocline location based on estimation of each node.

When a user needs to locate the thermocline, an initialization command is sent to any of the nodes. The node which receives the command begins to build the routing tree. When the tree is built, each node will explore the uppermost point and the lowermost point of its search regions, so that the user can send the query to find the thermocline. Any node which receives the query starts a local binary search and forwards the message to its children. Each node combines the reply from its children to send the report to its parent. This process is repeated, as every successive report has a better resolution on the thermocline location than the previous one. In the process, the nodes which are not located close to the thermocline stop to be active, and switch into the sleep mode to save power.

The system was tested on a testbed which consisted of 5 Mica2 nodes, one PC and one linear actuator with controller. The nodes were equipped with light sensors and thermistors. Each mote was able to talk to its neighbors over the radio, but no mote can talk to all other motes, because the communication range of the radio greatly was reduced underwater. The algorithm was implemented using TinyOS. The network was able to locate the thermocline correctly in most cases.

Additional experiments were conducted to use the robotic mote-based submarine as a data mule to reduce the number of needed message retransmissions. The results show that this approach is indeed useful. From a logistical point of view, it may be beneficial, as it is easier to replace the batteries in one data mule to conserve the battery power in many nodes.

In [7], [5], the authors describe a WSN to help vineyards managers to manage vineyards. The wireless sensor network consisting of 65 motes were in operation for 6 months at a vineyard in British Columbia. The motes were reporting temperature data towards a base station over multiple hops. Those data are needed to estimate heat unit requirements and cold hardiness. Every vineyard should have sufficient “Heat Summation Units” (HSU) or “Growing Degree Days” for the selected crop. This is a
Chapter 2. Characterization of WSN applications

characteristic of the temperature over 10°C (10°C is a baseline for wine grapes because they see no real growth until the temperature goes above 10°C). For example, HSU for Riesling are low, while those for Cabernet Sauvignon are high. So, in a cooler climate, even if there are not enough heat units for a Cabernet to mature, there may be enough for Riesling. This is significant because the per-ton price for Riesling is nearly half of what it is for Cabernet. If it would be possible to grow Cabernet, it would make financial sense to do so. Cold hardiness is the measure of exposure of the plant's resistance against the low temperatures. So, it makes sense to plant the harder plants in the areas of wineyards where the frosts are stronger. Another issue is to find the areas which have high risk of powder-mildew grape disease and to track areas which have been sprayed with pesticides [7].

The network in [5] was based on Mica-2 Berkeley motes and were using thermistors as sensors. To conserve power, the network was built with a two-tier multihop architecture, with a maximum of 8 hops. The first tier was composed of 16 motes that were 20% duty cycle and served as sensing motes and as primary routers for the network. The second tier was at 3% duty cycle and served as sensors only. The motes were deployed manually at a grid configuration. The location of nodes was planned so no ad-hoc routing was implemented. That allows to save some bandwidth and power at the time of network operation. The network took about one day to be deployed. Software was implemented using TinyOS. The base station was broadcasting a time synchronization signal at five minutes intervals. This also provided a simple mechanism to introduce new motes to the network or easily replace motes for maintenance. After synchronizing, the motes are sleeping for the majority of the time.

On average, data were received in 77% of cases. The strategy to resend each piece of data 5 times was significant, and allowed to get a complete data reports in many cases. The high cycle backbone required two battery changes during the deployment. The low duty cycle nodes showed no significant decline in power.

Another example of using sensor networks for edge detection is the LOFAR-argo project, run by researchers from Wageningen University and Delft University, Netherlands [39]. The purpose of the sensor network is to measure humidity and temperature in specific locations of the potato field. Obtained information about microclimate of the field can reveal when the crop is at risk of developing phitophthora. Phitophthora is a fungal disease which can spread among the plants and destroy a complete harvest within a large region. Early detection of the places where plants can develop a disease allows the farmer to treat the field, or parts of it, with fungicide only when absolutely needed. This precise treatment saves time, reduces costs, and limits the use of environment-unfriendly substances.
2.5 Tracking

In tracking, the source of an event can be mobile, e.g., an intruder in surveillance scenarios. The purpose of WSN tracking is to report updates on the event source’s position to the sink(s), potentially with estimates about speed and direction. To do so, typically sensor nodes have to cooperate before updates can be reported to the sink.

In [25], a tracking network is described. The application of the network is to alert military command and control unit about the presence of moving vehicles in a given region. The system should have extended lifetime, as surveillance mission could last from a few days to several months. It should be possible to adjust the network sensitivity to conserve power. The system should minimize its radio communication when no event is present to prevent the system from being detected or intercepted. The system is built using Mica2 motes platform, the used sensors are HMC1002 dual-axis magnetometers from Honeywell. Those sensors are omni-directional, have a resolution of 27 $\mu$Gauss, and are able to detect a small magnet at a distance of approximately 1 foot, and slowly moving passenger cars at a distance of approximately 8-10 feet.

The software is implemented in TinyOS, and consists of components providing time synchronization, localization, routing, data aggregation and power management. The operation of the system is divided into several phases. In the first phase, it performs the flooding of the network with messages to perform time synchronization, create network backbone, and make system-wide configuration. The second phase performs neighbor discovery: each node maintains the list of its neighbors. This information is used at the third phase to select sentry nodes. The sentry nodes are a subset of nodes which are responsible to awake other nodes when an event is detected. This allows the network to save power. After the routing backbone is created, at phase four, all nodes are sending status report to the sink for visualization and debugging purposes. This stage is optional. At phase five, nodes are performing power management and tracking. Power management is controlled by the sentry nodes, and consists of maintaining sleep-wakeup cycle. When an event occurs, nodes wake up and send reports containing the position of the node to the base station. If the nodes are densely deployed, multiple nodes may sense the event at the same time and send the same information to the sink, wasting bandwidth and energy.

To provide in-network aggregation, the nodes are organized into groups. Each group represents a separate event, and exists only as long as the event is in the scope of the sensor field. Each group is represented by a leader. Group members periodically report to the group leader. The leader records each report, keeping only the most recent one
from each member. Reports that are older than a certain threshold are deleted. The confidence level of an event detection is defined as the number of distinct nodes that had reported the event in the last period of time. When the confidence level of detecting an event is higher than the threshold, the leader sends a digest of the reports to the sink.

The network consisted of 70 nodes was deployed along 280 feet long perimeter in a grassy field. Empirical results have shown that the nodes were capable of tracking events. The in-network aggregation was essential to reduce retransmission overhead and provide a way to filter out false alarms. It was shown that several simplified assumptions which are usually made about the hardware platform and operating system do not hold in practice. The packet loss can be as large as 20%. So, it is important that the messages are retransmitted. But, to conserve power, only important messages should be retransmitted. The rate of false-alarm was non-negligible, as in-network aggregation was proved to be valuable. The race conditions which occur when different nodes try to transmit simultaneously or different components of the same node initiate transmission simultaneously may need a special attention when implementing real software for WSN. Communication between low power devices (like motes) is often asymmetric because of the differences in hardware, signal attenuation, and residual battery capacity. Then, special measures needed to take this into account. The same type of sensors in practice is capable of generating quite different sensor readings under identical conditions, due to differences in the way the devices are manufactured. As a result, additional calibration of the sensors in software may be needed.

In [3], authors describe ExScal, a large-scale surveillance network. The envisioned application is to detect, track and classify multiple intruders of different types (people, vehicles) to protect pipelines, borders, critical plants. The main requirements were low cost of covering a long perimeter over the mission lifetime; low false alarm and non-detected events rate; low human effort in deployment, maintenance and operation of network. Two types of hardware nodes were designed for this application: XSM (for Extreme Scale Mote) as a sensing node, and XSS (for extreme Scale Stargate) as a backbone communication node. XSM platform integrates an Atmel ATmega128L microcontroller, a Chipcon CC1000 radio operating at 433 MHz, a 4-Mbit flash memory, quad infrared, dual-axis magnetic, and acoustic sensors. A magnetometer is able to detect a vehicle at 7 meters. An infrared sensor is able to detect a vehicle at 30 meters and a person at 12 meters, and acoustic sensor is able to detect ATV (All terrain vehicle) at 50 meters. XSS includes a Linux-based Stargate computer, and a GPS unit which provides up to 10 m positioning accuracy. XSSs communicate via IEEE 802.11 network. External antenna allowed to achieve over 700 m reliable communication range.
Chapter 2. Characterization of WSN applications

ExScal uses a planned topology for node placement to cover the protected region. The application scenarios do not require sensor coverage in all points within the region, a barrier coverage is sufficient. So, sensors are deployed more densely at the boundary of the region than in its interior. At the outer border of the protected zone, there is a “thick” line of sensors which consists of 5 rows of XSMs. If any intruder crosses the thick line, it is detected by at least 5 sensors. So even if some intruder detection messages are lost due to network unreliability, classification and tracking of intruders are still possible. The interior of region consists of a grid of “thin” lines, each consisting of a single row of sensors. The topology ensures that each XSM can reliably communicate with 10 to 32 other XSMs in the thick line and 3-6 other XSM in the thin lines. Since the multi-hop reliability can be insufficient after 5-6 hops due to bandwidth constraints, the lines are divided into segments, where each 20-50 XSMs are provided with XSS which are relaying messages towards the base station. XSS are spaced 90 meters apart, a total of 45 XSS were needed to provide the communication backbone network. Thus, the network architecture is tiered - where XSM is a Tier 1, XSS - Tier 2, and base stations are Tier 3 nodes.

The software is divided into several applications: a Trusted Base program, a Deployment Application, a Localization Application and Perimeter security application. The Trusted Base program and the Deployment application are providing the way to change execution binary in every node, power management, information about the deployment status, node testing, network management and routing tree creation. The Localization Application assigns grid position labels to the nodes. The nodes which are equipped with GPS receivers are distributing the position information to the neighboring nodes, using the fact that nodes are deployed in a regular grid. The perimeter security application is combined with several services executed on three tiers. XSM detection packets from Tier 1 needs to be transported reliably to the XSS. Because the default distance-vector routing and queue management of TinyOS was achieving only 33.7% delivery rate, two special protocols for routing and transport are implemented. On Tier 2 network, there are three special services: initd service to initialize network, a structured convergecast service to transport information from any XSS to base station, and a third service is used to disseminate bulk data (up to 200 KB) to all XSS. The Tier 3 application logic implements the intruders classification. To calculate the influence field of each intruder, the detections are aggregated over time interval (usually 500 msec). Spatially collocated detections are clustered, then the cluster size is used for classification. The application maintains the history of decisions made in the recent past intervals to increase or decrease the confidence of classification. End-to-end classification of intruders is performed within 5 seconds. To track an intruder, the centroid of a convex region enveloping all the nodes detecting that intruder is calculated. Depending on the type of intruder, the tracking module also computes an expected region
for the intruder location in the next time interval. The tracked objects from successive windows are correlated to construct a continuous track per intruder for the entire time it spends in the network. If the estimated location of an intruder does not lie in the expected regions of any currently tracked intruders, a new intruder is detected. When there is no new information about a particular target within a certain time interval, the track is removed.

The system was deployed using more than 1000 XSMs and more than 200 XSSs in a 1.3 km by 200 m remote area in Florida. For realistic intruder scenarios, the end-to-end routing yield was 85.27%, the deployment faults were at 5.37%, localization faults at 11.4% and reprogramming faults were at 5.5%. Faults were uniformly distributed across the region. The overall Tier 1 and Tier 2 sensors reliability was 73%. This allows author to conclude that the designed system meets its requirements. It is believed that this design can be scaled to the network with a size about 10000 nodes.
Chapter 3

Node localization in wireless sensor networks

The performance of the WSN application (described in Chapter 2) requires that each node knows its location precisely. In this context, node localization in a wireless sensor network constitutes an active field of research [52], [59], [13], and [40]. This chapter presents an overview of the main methods for node localization, as well as the algorithms that implement such methods. It is organized into four sections. The first section presents the ranging methods. The second section describes the geometric positioning methods. The third section describes probabilistic positioning methods, whereas the fourth section presents some localization algorithms.

3.1 Ranging methods

*Ranging* is a process of measuring the distance between two WSN nodes. Information about this distance allows to calculate each node position. Several techniques are proposed to obtain this information [52]:

- **ToA** - by using time of arrival information;
- **TDoA** - by using time difference of arrival information;
- **RSSI** - by using received signal strength indicator.
3.1.1 Time of Arrival (ToA)

In the *Time of Arrival* (ToA) method, the traveling time of the signal is used to determine the distance between nodes. Imagine that the distance between two nodes \( A \) and \( B \) needs to be measured. At time \( t_0 \), node \( A \) sends a message to node \( B \). At time \( t_1 \), node \( B \) receives the message, and at time \( t_2 \) it sends another message back to node \( A \). The second message is received by node \( A \) at time \( t_3 \). Knowing the speed of propagation of the signal \( v \), the distance \( d \) between nodes \( A \) and \( B \) can be calculated using the formula 3.1:

\[
    d = \frac{v((t_3 - t_0) - (t_2 - t_1))}{2}
\]

(3.1)

where \( t_0 \) is the time when the first signal is sent, \( t_1 \) is the time when the first message is received, \( t_2 \) is the time when the second message is sent, and \( t_3 \) is the time when the second message is received. This procedure is illustrated in Figure 3.1. Systems using time of arrival method for ranging often use signals with slow speed of propagation, such as ultrasound. It enables to use less precise hardware to keep track of the time.

![Figure 3.1: Ranging using time of arrival (ToA) method](image)

3.1.2 Time Difference of Arrival (TDoA)

The *Time Difference of Arrival* (TDoA) method is similar to *Time of Arrival* method. However, it uses two signals with different speed of propagation: a fast-speed signal,
usually radio signal, and a slow-speed signal, usually ultrasound signal. At time $t_0$, node $A$ sends a radio signal. At time $t_1$, node $B$ receives the radio signal. After, at time $t_2$, node $A$ sends an ultrasound signal. At time $t_3$, node $B$ receives the ultrasound signal. Considering the speed of propagation of the radio signal $v_{rf}$ and the speed of propagation of the ultrasound signal $v_{us}$, the distance $d$ between nodes $A$ and $B$ can be calculated using formula 3.2:

$$d = \frac{(t_3 - t_1) - (t_2 - t_0)}{v_{rf} - v_{us}}$$

where $t_0$ is the time when the radio signal is sent, $t_1$ is the time when the radio signal is received, $t_2$ is the time when the ultrasound signal is sent, and $t_3$ is the time when the ultrasound signal is received. This procedure is illustrated in Figure 3.2.

![Figure 3.2: Ranging using time difference of arrival (TDoA) method](image)

### 3.1.3 Received Signal Strength Indicator (RSSI)

The Received Signal Strength Indicator (RSSI) method is based on the fact that the radio signal strength decreases with the distance. In this context, the path loss is the attenuation that a signal undergoes in travelling over a path between two points. The mathematical model for path loss can be described as follows [30]:

$$PL(d) = PL(d_0) + 10n\log\left(\frac{d}{d_0}\right)$$  (3.3)
where $PL(d)$ is the path loss function with respect to the distance measured in decibels, $PL(d_0)$ is the path loss over a reference distance measured close to transmitter, $n$ is the loss exponent which defines the rate at which the loss increases with the distance. This constant depends on the environment conditions, and is usually ranged from 2.0 to 5.0.

Using the signal strength to determine the distances usually yields to a number of errors [23], [8], [36], [81] because the actual path loss depends on many factors related to the environment, such as reflections, diffraction, scattering and antenna orientation. To model the error for the signal attenuation, a random variable is included in the path loss function:

$$PL(d) = PL(d_0) + 10n\log\left(\frac{d}{d_0}\right) + X_\rho$$

where $X_\rho$ is a zero-mean Gaussian random variable with a standard deviation $\rho$. In particular, it should be noted that choosing the proper probability distribution function to represent radio irregularities is not trivial. It should be verified that the real statistical data fit into the chosen distribution. Zhou et al. [81] suggest that the Weibull distribution may be better suited for this purpose. Weibull distribution is also well suited to model localization errors, which was demonstrated by Slijepcevic et al. [66].

### 3.2 Geometric positioning methods

*Positioning* is the process of determining the exact spatial coordinates of a WSN node. Geometric positioning methods are using geometric relations, such as distances or angles, to determine the positions. In this section, we present the main geometric positioning methods.

#### 3.2.1 Lateration

*Lateration* is used when there is a node for which we want to calculate the absolute position, knowing the absolute positions of several other nodes called *anchors*, as well as the distances to them. To localize a node in two dimensions, we need to know its distances to at least three anchors; to localize the node in three dimensions, we need to know the distances to at least four anchors.

For simplicity, assume the case of two dimensions. If we know the distances to exactly three anchors, we obtain a system of equations which has an unique solution.
This problem can be formalized as follows [27], [33]. Let \( v_i = (x_i, y_i) \), where \( i = 1, 2, 3 \) be a position of each anchor, \( v_u = (x_u, y_u) \) be the unknown position of the node to localize, and \( r_i, i = 1, \ldots, 3 \) be precise (with no error) distances between the unknown position and each anchor. This situation is illustrated in Figure 3.3.

From the Pythagorean theorem, a system of three equations is obtained as follows [27]:

\[
    r_i = \text{dist}(v_i, v_u) = \sqrt{(x_i - x_u)^2 + (y_i - y_u)^2}
\]

\[
    \begin{cases}
        (x_1 - x_u)^2 + (y_1 - y_u)^2 = r_1^2 \\
        (x_2 - x_u)^2 + (y_2 - y_u)^2 = r_2^2 \\
        (x_3 - x_u)^2 + (y_3 - y_u)^2 = r_3^2
    \end{cases}
\]

(3.6)

From (3.6), we obtain:

\[
    \begin{cases}
        (x_1 - x_u)^2 - (x_3 - x_u)^2 + (y_1 - y_u)^2 - (y_3 - y_u)^2 = r_1^2 - r_3^2 \\
        (x_2 - x_u)^2 - (x_3 - x_u)^2 + (y_2 - y_u)^2 - (y_3 - y_u)^2 = r_2^2 - r_3^2
    \end{cases}
\]

(3.7)
which corresponds to:
\[
\begin{aligned}
(x_1^2 &- 2x_1x_u + x_u^2) - (x_3^2 - 2x_3x_u + x_u^2) + (y_1^2 - 2y_1y_u + y_u^2) - (y_3^2 - 2y_3y_u + y_u^2) = r_1^2 - r_3^2 \\
(x_2^2 &- 2x_2x_u + x_u^2) - (x_3^2 - 2x_3x_u + x_u^2) + (y_2^2 - 2y_2y_u + y_u^2) - (y_3^2 - 2y_3y_u + y_u^2) = r_2^2 - r_3^2
\end{aligned}
\] (3.8)

Rearranging of terms results in:
\[
\begin{aligned}
2(x_3 - x_1)x_u + 2(y_3 - y_1)y_u &= (r_1^2 - r_3^2) - (x_1^2 - x_3^2) - (y_1^2 - y_3^2) \\
2(x_3 - x_2)x_u + 2(y_3 - y_2)y_u &= (r_2^2 - r_3^2) - (x_2^2 - x_3^2) - (y_2^2 - y_3^2)
\end{aligned}
\] (3.9)

which can be rewritten in a matrix form as follows:
\[
2 \begin{bmatrix}
x_3 - x_1 \\
x_3 - x_2 \\
\vdots \\
x_3 - x_{n-1}
\end{bmatrix} \begin{bmatrix}
y_3 - y_1 \\
y_3 - 2y_2 \\
\vdots \\
y_3 - 2y_{n-1}
\end{bmatrix} \begin{bmatrix}
x_u \\
y_u
\end{bmatrix} = \begin{bmatrix}
(r_1^2 - r_3^2) - (x_1^2 - x_3^2) - (y_1^2 - y_3^2) \\
(r_2^2 - r_3^2) - (x_2^2 - x_3^2) - (y_2^2 - y_3^2) \\
\vdots \\
(r_n^2 - r_{n-1}^2) - (x_n^2 - x_{n-1}^2) - (y_n^2 - y_{n-1}^2)
\end{bmatrix}
\] (3.10)

In real life, the range measurements are not perfect and contain some errors. To take this into account, we represent the imprecise range measurement in the form \( \tilde{r}_i = r_i + \epsilon_i \), where \( r_i \) represents a perfect measurement, and \( \epsilon_i \) is an unknown measurement error. In this case, solving the system for trilateration does not always give an exact solution. In fact, it is possible that three circles are intersecting in more than three points. This is illustrated in Figure 3.4. In this case, more than three anchors are used, resulting in an overdetermined system of equations to take into account individual measurement errors [27]. This overdetermined system of equations can be written in a matrix form as follows:
\[
2 \begin{bmatrix}
x_n - x_1 \\
x_n - x_2 \\
\vdots \\
x_n - x_{n-1}
\end{bmatrix} \begin{bmatrix}
y_n - y_1 \\
y_n - 2y_2 \\
\vdots \\
y_n - 2y_{n-1}
\end{bmatrix} \begin{bmatrix}
x_u \\
y_u
\end{bmatrix} = \begin{bmatrix}
(r_1^2 - \tilde{r}_n^2) - (x_1^2 - x_n^2) - (y_1^2 - y_n^2) \\
(r_2^2 - \tilde{r}_n^2) - (x_2^2 - x_n^2) - (y_2^2 - y_n^2) \\
\vdots \\
(r_n^2 - \tilde{r}_{n-1}^2) - (x_n^2 - x_{n-1}^2) - (y_n^2 - y_{n-1}^2)
\end{bmatrix}
\] (3.11)

For this overdetermined system of equations, a solution can be computed by minimizing the mean square error \( \| Ax - b \|^2 \), where \( 0.5A \) is the left-hand matrix, \( x = (x_u, y_u) \) is a vector describing the unknown position of the node, and \( b \) is the right-hand side vector of equation (3.11). The term \( \| Ax - b \|^2 \) is the Euclidean norm squared of the residual \( Ax - b \), which is equal to:
\[
\| Ax - b \|^2 = (Ax - b)^T (Ax - b) = x^T A^T Ax - 2x^T A^T b + b^T b
\] (3.12)
where \((Ax - b)^T\) is the matrix transpose of \((Ax - b)\).

Regarding (3.12) as a function of \(x\), minimizing this expression means that the gradient of this function should be set equal to zero [27], which leads to:

\[
2A^T Ax - 2A^T b = 0
\]  

(3.13)

This is equivalent to:

\[
A^T Ax = A^T b
\]  

(3.14)

This equation is called the normal equation for the linear least square problem [27]. If \(A\) has a full column rank, the solution of this equation is unique and given by:

\[
x = (A^T A)^{-1} A^T b
\]  

(3.15)

where \((A^T A)^{-1}\) is the matrix inversion of \((A^T A)\). Karalar et al. [33] discuss the implementation of custom-made hardware usable for a WSN node to solve this system.

### 3.2.2 Angulation

If it is possible to estimate the angle towards the anchors, basic geometric laws can be used to determine the node positions. This approach is called *angulation* and illustrated by the diagram.

![Figure 3.4: Using multilateration for position evaluation based on imprecise ranging measurements](image)
in Figure 3.5. To localize the node on a plane, it is sufficient to know two angles towards it from two different known positions. In practice, angulation is difficult to use for WSN applications because of the additional hardware needed to measure the angles. This hardware usually involves antenna arrays [74], [52], or rotating beacons [49], which is often prohibitive for the size and power-constrained WSN nodes.

\[ (x_1, y_1) \]
\[ (x_2, y_2) \]
\[ (x_u, y_u) \]

Figure 3.5: Using triangulation to determine a node position

### 3.2.3 Multi-Dimensional Scaling

*Multi-Dimensional Scaling* (MDS) is a method which has its origin from psychometrics and psychophysics [11]. It is a set of data analysis techniques that display the structure of distance-like data as a geometrical picture. MDS starts with one or more distance matrices (or similarity matrices) which represent the points in a multidimensional space. It is usually used to find the placement of the points in a low-dimensional space, usually two or three-dimensional, in a way that preserves distances between points in original, multidimensional space. MDS is often used as part of exploratory data analysis or information visualization. It allows to visualize objects as points in a low-dimensional space, reducing the complexity in the original data matrix while preserving the essential information. Applications of this method for WSN node localization are discussed in [11], [16], [63].

The input of the method is the distance estimations between each pair of nodes. From this estimation, using the MDS, it is possible to derive the relative position of
each node, which preserves initial distance constraints, up to arbitrary rotation [16].
If the absolute locations of some anchors are known, to obtain the absolute location of the nodes, additional transformation is performed. More specifically, assume that there are \( N \) points with coordinates \( x_i \), where \( i \in 1..N \). It is possible to measure the pairwise dissimilarities \( \{\delta_{ij}\}_{i,j=1}^N \), such as dissimilarities corresponding to the true Euclidean distances [16]:

\[
\delta_{ij} = d_{ij} = d(x_i, x_j) = \|x_i - x_j\| = \sqrt{(x_i - x_j)^T(x_i - x_j)} \tag{3.16}
\]

where \((x_i - x_j)^T\) is the vector transpose of \((x_i - x_j)\). Taking into account that

\[
d_{ij}^2 = x_i^T x_j - 2x_i x_j + x_j^T x_j \tag{3.17}
\]

the squared distance matrix \( D = [d_{ij}^2]_{i,j=1}^N \) can be written as:

\[
D = \psi e^T - 2X^T X + e\psi^T, \tag{3.18}
\]

where \( e = [1, \ldots, 1] \) is the \( N \)-dimensional vector of all ones, and \( \psi = [x_1^T x_1, \ldots, x_N^T x_N]^T \).

Then, we compute the centering matrix \( H \) as follows:

\[
H = I - \frac{1}{N} ee^T \tag{3.19}
\]

where \( I \) is the identity matrix of size \( N \). Then, applying double centering to \( D \) enables to obtain a scalar product matrix \( B \) as follows:

\[
B = -HDH = HX^T XH \tag{3.20}
\]

Then we calculate eigen decomposition of matrix \( B \) [11]:

\[
B = UVU^T \tag{3.21}
\]

where \( V = \text{diag}(\lambda_1, \ldots, \lambda_N) \) is the diagonal matrix of eigenvalues of \( B \) with \( \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_N \geq 0 \), and \( U = [u_1 u_2 \ldots u_N] \) is an orthonormal matrix whose columns are their corresponding eigenvectors.

After multiplication with \( H \), the columns \( X^T X \) have a zero mean. Now, given \( B \), matrix \( X \) can be recovered up to a translation and orthogonal transformation, as the solution to the following variational problem:

\[
\min_Y \|B - Y^T Y\|_F^2 \tag{3.22}
\]

where \( \| \cdot \|_F \) is the Frobenius norm, and the minimum is taken over all \( D \times N \) rank-\( D \) matrices. The solution of (3.22) is given by:

\[
X = \text{diag}(\lambda_1^{1/2}, \ldots, \lambda_K^{1/2}) U^T, \tag{3.23}
\]

where \( K \) is the number of dimensions we are interested in (usually 2 or 3). There is little overhead for computing the coordinates in 3D space as compared to 2D space, which is a nice property that other positioning methods do not have [62].
3.3 Probabilistic positioning methods

Probabilistic positioning methods use the theory of probability to deal with ranging errors and missing information. These methods include location fingerprinting and Monte Carlo localization.

3.3.1 Location fingerprinting

Location fingerprinting (also called “scene analysis”) methods work by assigning to each location some measurable characteristics. Articles [4], [32] [79] describe using radio signal characteristics, such as RSSI or signal to noise ratio, as “fingerprints”. Radio-frequency identification (RFID) tags [31], or even visible light [57] can be used as well. Scene analysis methods involve two phases: an off-line phase and a real-time phase. When the network is deployed, at the off-line phase, a central database is filled with the “fingerprints” of each location [4].

At the real-time phase, when any node needs to localize itself, it queries the database to determine its location. To determine which fingerprint from the database corresponds to the received signal, the nearest neighbor in signal space (NNSS) approach is used. This approach calculates the Euclidean distance in signal space between the received signal and each record in the database. The location with the minimum distance is chosen as the estimated location of the localized node. Extension of this algorithm is NNSS-AVG, which tries to pick several training locations which are close to the received signal. Then, the position of the localized node is inferred based on the selected training locations.

3.3.2 Monte Carlo localization

Monte Carlo localization method is a probabilistic approach for mobile node positioning. This method is traditionally used in robotics. The localization problem for Monte Carlo method may be formulated as in [18]. Let \( t \) be the discrete time variable; \( l_t \) is the position distribution of a node at time \( t \); \( o_t \) represents the observation of a node received from the anchors between time \( t - 1 \) and \( t \). The method recursively estimates in time the so-called filtering distribution: \( P(l_t|o_{0:t}) \). The filtering distribution is represented as \( N \) weighted samples which are updated every time unit, using an importance function [18]. In the ideal case, samples are drawn from the posterior distribution: \( P(l_{0:t}|o_{0:t}) \).
But, most of the time, it is impossible to sample from the posterior distribution. The general Monte Carlo localization method is expressed as follows:

$$P(l_{0:t}|o_{0:t}) = P(l_0|o_0) \prod_{k=1}^{t} P(l_k|l_{0:k-1}, o_{0:k})$$

(Article [29] proposes a method which does not use any range measurements, and only relies on connectivity constraints and the assumption that the speed of the moving nodes does not exceed some maximum value $v_{\text{max}}$. This method is summarized in Algorithm 1.

**Algorithm 1 Monte Carlo Localization**

**Initialization:** Initially the node has no knowledge of its location $N$ is a constant that denotes the number of samples to maintain

$L_0 = \{ \text{set of } N \text{ random locations in the deployment area} \}$

**Step:** Compute a new possible location set $L_t$ based on $L_{t-1}$, the possible location set from the previous time step, and the new observations, $o_t$.

$$L_t = \{\}$$

while $\text{size}(L_t) < N$ do

$R = \{l_i^t, l_i^t \text{ is selected from } p(l_t|l_{t-1}^{i-1}), l_{t-1}^{i-1} \in L_{t-1} \text{ for all } 1 \leq i \leq N\} \triangleright \text{Prediction}$

$R_{\text{filtered}} = \{l_i^t, l_i^t \text{ where } l_i^t \in R \text{ and } p(o_t|l_i^t) > 0\} \triangleright \text{Filtering}$

$L_t = \text{choose}(L_t \cup R_{\text{filtered}}, N)$

end while

The first phase of each step is a prediction phase. A transition equation $p(l_t|l_{t-1})$ describes the prediction of the node’s current position based on the previous position. A node starts from the set of possible locations $L_{t-1}$ computed in the previous step and applies the mobility model to each sample to get a set of new samples $L_t$. It is assumed that the node is unaware of its moving speed and direction, but it knows that its speed cannot exceed $v_{\text{max}}$. So, if in the previous step $l_{t-1}^i$ is one of the possible positions of the node, the possible current position is contained within the circular region with origin $l_{t-1}^i$ and radius $v_{\text{max}}$. If $d(l_1, l_2)$ is the Euclidean distance between two points $l_1$ and $l_2$, and the speeds are distributed uniformly in the interval $(0, v_{\text{max}})$, a transition equation is given by the following uniform distribution:

$$p(l_t|l_{t-1}) = \begin{cases} \frac{1}{\pi v_{\text{max}}}, & \text{if } d(l_t, l_{t-1}) < v_{\text{max}} \\ 0, & \text{if } d(l_t, l_{t-1}) \geq v_{\text{max}} \end{cases}$$

At the second phase (filtering), the node filters out the impossible locations based on new observations. Let $S$ be the set of all anchors heard by the node, and $T$ the set
of the anchors heard by the node’s neighbors, but not by the node. Then, the filter condition of location \( l \) is:

\[
\text{filter}(l) = \forall s \in S, d(l, s) \leq r \land \forall s \in T, r < d(l, s) \leq 2r
\]  

(3.26)

The probability distribution \( p(l_t | o_t) \) is zero if the filter condition is false, and evenly distributed otherwise. Thus, we eliminate locations that are inconsistent with observations from the possible location set. After filtering, there may be fewer than \( N \) possible locations remaining. The prediction and filtering are repeated, accumulating the possible points found, until at least \( N \) possible locations have been acquired.

This method can be extended to take into account the range measurements, as discussed in [18]. Another approach which uses Monte Carlo localization together with RSSI-based location fingerprinting database is discussed in [79].

### 3.4 Existing localization algorithms

Because of the importance of the localization problem many solutions were proposed [40], [53]. These solutions can be grouped into several categories, depending on the applications, the assumptions made and the resources available. This section presents a categorization of existing localization algorithms, as well as the most important algorithms.

#### 3.4.1 Categorization

Some algorithms, called anchor-based algorithms [2], [46], [31], are relying on the presence of nodes which know their precise position before the localization algorithm starts. Those nodes are called anchors. The presence of anchors enables to decrease the computational complexity of calculations and allows to obtain the coordinates which are in accordance with some global coordinate system. At the same time, anchors maintenance and deployment in some situations are not possible. So, another set of algorithms, called anchor-free algorithms [47], [6] attempt to find the motes position when no anchor is available.

Anchor-based algorithms can be further subdivided into two groups: single-hop and multi-hop algorithms. Single-hop algorithms assume that every non-anchor node of the
WSN at any moment can directly communicate with enough anchors to localize itself [78]. In practice, it means either that anchor nodes have more powerful transmitter to cover the whole network, or there is sufficient density of anchor motes in the WSN deployment. In *multi-hop algorithms*, position information is propagated via the network, and non-anchor nodes are using also position information from their non-anchor neighbors to localize themselves [60], [61]. Single-hop algorithms generally require networks with more connectivity, whereas multi-hop algorithms support more sparse networks at the cost of decreased precision.

Some algorithms assume that the network nodes are able to estimate or measure the distance to each other [33], [46],[70]. These algorithms are called *range-based algorithms*. Explicit distance measuring allows to achieve greater precision of localization. This comes with the price that additional hardware is often needed to estimate the range. Range-based algorithms can be further subdivided into classes, based on which method of range measurement is used. *Range-free algorithms* are trying to rely on other sources of information for positioning [2], [43].

*Centralized algorithms* use network nodes to collect data for localization computation, which runs on a main central node. In *distributed algorithms*, no central node exists, as all nodes collaborate to localize each other. Having a central node in a system allows to overcome resource limitation of the sensor nodes, so that more computationally demanding algorithms can be used. At the same time, a single central node introduces a point of failure for the whole system, while reducing the scalability of the system. For large networks, a central node means much communication overhead, increasing latency, and demand for more battery power.

In [52], Patwari et al. discuss unavoidable limits to localization accuracy for anchor-based, multi-hop, range-based localization algorithms. Range and angle measurements used for localization are performed in a physical medium that introduces errors. Measurements-based statistical models for RSSI, ToA and AoA measurements are presented. These models allow to calculate a lower bound on sensor location estimation variance. This calculation is based on statistical result called Cramér-Rao bound (CRB). The *Cramér-Rao bound* provides a lower bound on the variance achievable by any unbiased location estimator. It is possible to calculate the lower bound on estimation variance without ever considering a single estimation method. The bound on estimator covariance is a function of the number of anchor and non-anchor nodes, sensor geometry, whether localization is in two or three dimensions, measurement type and its precision, network connectivity, nuisance (unknown) parameters that must also be estimated. Using the presented statistical models of measurements, the scaling characteristics of variance are explored. It is shown that ToA bounds will remain constant.
with a dimensions scaling. If units of the coordinates are in feet instead of meters, the error ratios will stay the same. At the same time, the bounds on localization error for ToA and RSSI are proportional to the size of the system. An RSSI-based localization system operating in a high-path-loss exponent environment while requiring higher transmission power from sensors will produce more accurate sensor localization.

### 3.4.2 Ad hoc Positioning System

The Ad hoc Positioning System (APS), proposed by Niculescu et al. [50], is an anchor-based, multi-hop algorithm which focuses on the assumption that, at the beginning of the localization, only a limited number of anchors is available, and no non-anchor node can receive beacons from the three anchors, so direct lateration is not possible. Immediate neighbors of the anchors are determining their distance to the anchors and to their neighbors. Once a node obtains a range estimate to more than 3 anchors, it can determine its position by lateration and become an anchor. Thus, the location estimate is propagated from the anchors to the middle of the network.

Three propagation methods are proposed: “DV-hop”, “DV-distance” and “Euclidean”. The “DV-hop” propagation method is a basic scheme which consists of three stages. First, all nodes are obtaining the hop-count distance to the anchors by direct propagation. Each node maintains a table $X_i, Y_i, h_i$, where $(X_i, Y_i)$ are coordinates of the anchor $i$ and $h_i$ is a hop-count. The nodes exchange updates only with their neighbors. Second, each anchor collects the hop count to other anchors, estimates the average length of the hop, and sends this information as correction to the nodes in its neighborhood. Third, when receiving the correction, a non-anchor node may estimate the distance to the anchor, and perform lateration. The correction for the node with coordinates $(X_i, Y_i)$ is calculated as follows:

$$c_i = \frac{\sum \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2}}{\sum h_i}, \quad i \neq j, \quad \text{all anchors } j.$$  

(3.27)

The “DV-distance” propagation method is similar to the “DV-Hop” with the difference that the distance between neighboring nodes is measured using some ranging method. The “Euclidean” propagation method is trying to calculate true Euclidean distance to an anchor. For that, a non-anchor node $A$ needs to have at least two neighbors $B$ and $C$ who estimate their distances to an anchor $L$. $A$ also has measured estimates of distances for $AB$, $AC$, and $BC$, so there is the condition that: either $B$ and $C$, besides being neighbors of $A$, are neighbors of each other, or $A$ knows distance $BC$, from being able to map all its neighbors in a local coordinate system. In any case, for the quadrilateral $ABCL$, all the sides are known, and one of the diagonals, $BC$, is also
known. This allows node $A$ to compute the second diagonal $AL$, which in fact is the Euclidean distance from $A$ to the anchor $L$.

The APS implementation is described in [77]. The system was implemented using Mica2Dot motes, as the ultrasonic TDoA method was used for ranging. The deployment involved 49 nodes on a paved surface over an area of 13 by 13 meters. There were 4 anchors. A median location error of about 47 cm is reported.

### 3.4.3 Anchor-Free Localization

Anchor-Free Localization (AFL) was proposed by Priyantha et al. [55]. The algorithm is fully distributed and consists of two phases. The first phase generates the fold-freedom representation of the distance graph, as illustrated in Figure 3.6.

![Figure 3.6: Graph rigidity: 1 - non-rigid graph; 2 - rigid but not globally rigid graph; 3 - globally rigid graph](image)

To find the rigid representation of the distance graph, simple heuristics are used. In a distributed manner, five reference nodes are chosen in a way that 4 nodes, $n_1, n_2, n_3$ and $n_4$ at the each side of the network and one, $n_5$ - roughly in the middle of the network, and line between $(n_1, n_2)$ is roughly perpendicular to the line between $(n_3, n_4)$. This is done using the hop-count information as a distance metric.
Chapter 3. Node localization in wireless sensor networks

For each node $i$, its polar coordinates $(\rho_i, \theta_i)$ are calculated as:

$$
\rho_i = h_{5,i} \times R \\
\theta_i = \arctan\left(\frac{h_{1,i} - h_{2,i}}{h_{3,i} - h_{4,i}}\right)
$$

where $R$ is the communication range, and $h_{i,j}$ is a number of hop-counts along the shortest path between nodes $i$ and $j$.

The second phase uses a force-based relaxation procedure to find precise locations. To measure the performance of the solution, a global error ratio (GER) is proposed as follows:

$$
GER = \sqrt{\frac{\sum_{i,j:i<j} \hat{e}_{ij}^2}{N(N-1)/2}}
$$

where $\hat{e}_{ij}$ is a normalized error in a distance measurement between two nodes $i$ and $j$.

Simulation results report that the GER is less than 0.0012 if the ranging error does not exceed 7%. It looks like the localization precision depends on the network connectivity more than on the ranging precision. The maximum error between any two unconnected nodes is up to $0.8R$ if the connectivity is 13 and about $3R$ if the connectivity is 5.

3.4.4 MoteTrack

MoteTrack is an anchor-based, range-free distributed algorithm which uses RSSI signatures for location fingerprinting [44]. In the MoteTrack algorithm, anchor nodes broadcast beacon messages at various transmission power levels. Each beacon message has the form $(sourceID, powerLevel)$, where sourceID carries the ID of the beacon mote who sent it, and powerLevel contains the transmission power level used to broadcast the message. Each non-anchor node that wants to determine its location aggregates beacon messages received over some time interval into a signature, which is then compared with a reference signature with a known location. Each beacon node stores a subset of all reference signatures.

The MoteTrack algorithm has two phases: an offline collection of reference signatures and an online location estimation. The reference signature database is built manually, and consists of a set of signature tuples in the form $(sourceID, powerLevel, meanRSSI)$, where sourceID is the anchor node ID, powerLevel is the transmission power level of the beacon message, and meanRSSI is the mean RSSI of a set of beacon
messages received over some time interval. Each signature is manually mapped to a known location at the time when the signature database is constructed.

When a non-anchor node receives a signature $s$, having the reference signature set $R$, the node location is calculated as follows. First, the signature distance from $s$ to each reference signature $r_i \in R$ is calculated as follows:

$$M(r, s) = \sum_{t \in T} |\text{meanRSSI}(t)_r - \text{meanRSSI}(t)_s|$$

(3.31)

where $T$ is the set of signature tuples presented in both signatures, $\text{meanRSSI}(t)_r$ is the mean RSSI value in the signature tuple $t$ appearing in signature $r$. Then, the location of the non-anchor node is calculated as the centroid of the set of signatures within some ratio of the nearest reference signature. Given a signature $s$, a set of reference signatures $R$, and the nearest signature $r^* = \arg\min_{r \in R} M(r, s)$, the algorithm selects all reference signatures $r \in R$ that satisfy:

$$\frac{M(r, s)}{M(r^*, s)} < c$$

(3.32)

where $c$ is a constant. Second, the geographic centroid of the locations of this subset of reference signatures is then taken as the non-anchor node’s position.

The algorithm is designed to gracefully handle incomplete data and failed nodes. The reference signature database is replicated across the beacon nodes, so that each beacon node stores a part of it. An adaptive algorithm for the signature distance metric is used to take into account partial failure of the anchor nodes infrastructures. Each anchor node dynamically estimates the current fraction of locally failed anchor nodes and switches to a different distance metric to offset location errors caused by such failures.

The MoteTrack was implemented using Mica2 motes using TinyOS operating system, and deployed over one floor of a building, measuring roughly $1742 \text{ m}^2$. The installation consisted of 20 anchor motes. A total of 482 reference signatures were collected. Each signature was collected for 1 minute, during which time every anchor mote transmitted at a rate of 4 Hz, each cycling through 7 transmission power levels. It is noted that collecting references for several seconds for each signature may be sufficient. The accuracy within $3 \text{ m}$ is reported for 80% of tests measures.
Chapter 4

Implementation procedures, experiments and results

In order to produce precise mote locations, the majority of localization algorithms implement range-based methods for distance evaluation [33], [61], [47]. In this context, one of the most used methods is RSSI [65], [64], [2]. This chapter describes experiments that were made for RSSI measurements, and presents the analysis of the results obtained from these experiments. It is organized into three sections. The first section describes the experiment design. The second section describes the implementation of the WSN application used to collect RSSI data. The third section presents experiments which were run, and analyses the set of collected data.

4.1 Experiment design

The experiment design defines which questions the experiment results are supposed to answer, as well as the characteristics and properties of the data to be collected, and the procedure used for obtaining such data. In this section we describe our hypotheses about RSSI measurements and outline the experiments used to collect real data to verify them.
4.1.1 Goals and hypotheses

The goal of our experiments is to collect data to allow us to understand how RSSI measurements could be used for measuring the distance between two WSN nodes. What would be the transmission range for real devices running in real environments? Do RSSI measurements vary for different devices in the same conditions? Is RSSI different for the same distance at different environments? If it is, what may affect those differences? In this case, what is the variation in RSSI measurements? Are there some grey areas where RSSI measurements have higher variation that could be detected?

Answers to all these questions affect the choice of the algorithm, requirements and performance of the localization application. If RSSI measurements do not vary significantly with environment, it would be possible to hardcode the dependence between RSSI and distance inside the WSN motes. Otherwise, the localization application needs to be calibrated for each specific deployment. If the RSSI signal varies significantly for the same distance, the precision of using RSSI for ranging would be low. In this case, it would be preferable to use probability-based or signature-based methods of localization instead of geometry-based methods.

Before running the experiments, we would start with a couple of hypotheses based on intuition and data from previous research to guide our tests [26], [28]:

- RSSI varies significantly and depends on the environment;
- RSSI is mostly stable over time;
- Measuring RSSI is mostly reliable from device to device;
- It is possible to use signals of different transmission power to improve the precision of localization.

4.1.2 Preliminary concepts

Before the measurements start, two or more motes are put in predefined positions. Distances between the motes, as well as their coordinates are measured with a measuring tape. This stage of the experiment is called deployment. One of the motes starts broadcasting radio signals called beacons. When these signals are received by the other motes, RSSI values are measured and logged. The mote which sends the beacons is called a sender. The motes which measure RSSI levels are called receivers. The sender
Chapter 4. Implementation procedures, experiments and results

sends the predefined number of beacons with the given transmitted power one after another with some delay between them. When the sender has finished sending beacons, data from each receiver are transferred to the host computer. This part of experiment, consisting of one sender sending beacons and transferring measured RSSI samples from all receivers, is called trial. When a trial finishes, another mote may become the sender and the next trial starts. Several experiments are run, using different parameters.

4.2 Implementation of the WSN application

This section describes design and implementation of the WSN application used for collecting RSSI data. More specifically, it explains the hardware and software platform used for implementation, application requirements, and architecture, as well as the communication protocol implemented for reliable communication with WSN motes.

4.2.1 Hardware and software platform

For our experiments, we are using Tmote Sky wireless sensor modules from Moteiv (www.moteiv.com). Each module is controlled by Texas Instruments MSP430 microcontroller which has 10 kB RAM, 48 kB Flash memory, and runs at 8MHz [48]. Also, each module provides integrated light, temperature and humidity sensors. For radio communication, this module uses Chipcon CC2400 RF transceiver chip. This chip is compliant with IEEE 802.15.4 standard and allows to measure received signal strength which can be read from \textit{RSSI.RSSI\_VAL[7 : 0]} register.

According to [15], the RSSI reading provides a measure of the signal power entering the RF input. The scale is logarithmic, so that \textit{RSSI\_VAL} provides a value in dBm. The RSSI measurement can be referred to the power at the RF input pins by using the following equation:

\[
P = \text{RSSI\_VAL} + \text{RSSI\_OFFSET} \text{ [dBm]} \tag{4.1}
\]

The nominal value of \textit{RSSI\_OFFSET} is -54 dBm. When presenting the experiment results, we will present the direct measurements, without applying offsets.

The firmware is running on a Tmote Sky mote, controlled by TinyOS [71]. TinyOS
is a popular and widely used operating system for WSN systems. It provides hardware abstraction for a wide range of commercially available sensor nodes, implements several communication protocols and provides the library of reusable components for developing WSN applications. TinyOS is implemented in nesC [22], a special language which is designed for sensor node programming. More specifically, nesC is designed to facilitate writing programs which are driven by the interaction with the environment, have limited resources and should be reliable.

4.2.2 Requirements and deployment

The requirements for the WSN application are specified based on the planned experiments design. First, it should be possible to change the experiment parameters - transmission power, number and coordinates of motes, number of beacons sent by each sender. Second, deployments are expensive - it is needed to place motes in specific positions and measure distances, so that it should be possible to download all collected samples or be able to detect missing data. Third, radio communications are unreliable and some motes may become unaccessible during the experiments (in case of battery failure). Thus the application should be robust against missing messages and single mote failure - data from the working motes should be downloaded, and the experiment should be able to continue with the working motes.

A WSN application is a distributed program which consists of several modules which are executed on different computers. These modules are illustrated at the deployment diagram in Figure 4.1. The deployment diagram nodes represent different software packages, and the lines represent the data flow between them.

The application execution is controlled by a host computer. The host computer is connected with a gateway mote with a USB cable. It runs two modules: the serial forwarder application and the experiment control application. The serial forwarder is an application written in Java provided as a part of Tmote Sky motes distribution. It reads data from the gateway mote via USB, detects when a complete message from a WSN mote is available and forwards it to any other application using TCP socket. The experiment control script is written in python.

The gateway mote connected to the host computer executes a standard base station application which is a part of TinyOS distribution. This application listens to the radio, and when a message from the radio is received, this message is resent via a serial interface, so that it becomes available to the serial forwarder.
Each other WSN mote executes an experiment application. There is only one kind of experiment application which allows a mote to play the role of sender or receiver, depending on the commands received from the application control script. Each mote has a unique address. The experiment control script knows the addresses of all motes in the network. There is no multi-hop communication, every mote sends messages directly to the gateway mote.

### 4.2.3 Application behavior

The application behavior is represented by the activity diagram illustrated in Figure 4.2. The nodes of the activity diagram represent operations performed by the application, whereas the lines represent the control flow of the application. Logically, the application operation can be divided into the following stages:

1. At the *initialization stage*, the experiment control script loads a scenario file, and then establishes a connection to communicate with the gateway mote.

2. At the *deployment stage*, the experiment control script displays the mote coordinates at the screen, and waits for the user to place the motes in a proper location and to press the “Start” button.

3. Based on the scenario file, the control script chooses one of the motes as a sender
mote. All the other motes are commanded to become receivers. The command is sent to the sender mote to start sending beacons, starting the measuring stage.

4. A message from the sender mote informs the experiment control script that sending of all beacons is finished. The experiment control script loops through each receiver mote sending the transfer request, starting the transfer stage. On reception of the transfer request, each receiver mote starts sending all collected RSSI data to the gateway mote. During the transfer the transmission power level is set to maximum. Each transfer message holds a message number, so missed packets are detected and retransmitted.

![Activity diagram](image)

**Figure 4.2: Activity diagram**

Measuring and transfer stages are repeated for each deployment accordingly to the experiment scenario. The experiment parameters are set up using an XML scenario file as follows:

```xml
<scenario>
  <deployment>
    <trial beacons_number="6" tx_power="30" delay="900"/>
    <mote id="1" x="0" y="0" z="0"/>
    <mote id="2" x="1" y="0" z="0"/>
  </deployment>
  <deployment>
    <trial beacons_number="6" tx_power="30" delay="900"/>
  </deployment>
</scenario>
```
Each deployment is specified by the number and coordinates of motes, as well as the number and parameters of the trials. Each mote has a network address and its coordinates. Each trial specifies the number of beacons to be sent by the sender mote, as well as the number, delay duration between beacons and transmitted power.

The logic of the experiment may be expressed as the following pseudocode:

```python
scenario = scenario.load('scenario.xml')
for d in scenario.deployments(){
    for t in d.trials(){
        for s in t.motes(){
            for r in t.receivers(s){
                r.start_beacon(s, t.params)
            }
            s.start_beacon(s,t.params){
            }
            for r in t.receivers(s){
                r.start_transfer()
            }
        }
    }
}
```

This pseudocode describes the application logic in a procedural manner, as if it is executed by a single computer sequentially. In reality, an application logic is implemented via collaboration of many motes and the host computer. Several modules of the WSN application are executed in parallel, exchanging information via messages. Thus, the real behavior of each module is reactive, whereas the computations performed by each mote is specified as a response to external and internal events.

There are 9 types of messages exchanged between the motes and the host computer. They are discussed below, grouped by the functionality they provide to the application. Among them, 3 types are used to allow the application to measure RSSI values. They are: `BeaconStartMsg`, `BeaconMsg` and `BeaconFinishedMsg`. 
BeaconStartMsg(sender, beacons_number, tx_power, delay) is sent by the experiment control script to start the measurement stage. On reception of this message, the mote compares the sender field with its own network address. If the sender field of the message is equal to the mote’s network address, the mote becomes a sender mote and prepares to send beacons_number beacons with a pause between beacons equal to delay milliseconds, while setting the transmission power level for each beacon to the value of tx_power message field. If the sender field of the message does not match the mote’s address, the mote becomes a receiver and starts waiting for beacons. BeaconMsg(sender, counter, tx_power) is broadcasted from the sender mote to the receiver motes. On reception of this message by a receiver mote, RSSI is measured. The sender field carries the network address of the sender mote. The counter field is used to distinguish one beacon from another: it is incremented from 0 to (beacons_number - 1) during the trial. The tx_power field carries the transmission power level used during beacon sending. BeaconFinishedMsg is sent by the sender mote to the host computer to notify that all the beacons were sent.

Also, 3 types of messages are needed to allow the application to collect measured RSSI data. They are: TransferStartMsg, TransferMsg and TransferFinishedMsg. TransferStartMsg is sent by the host computer to the receiver mote to start collecting measured RSSI data. TransferMsg is sent by the receiver mote to the host computer. It has fields receiver_counter, rssi, lqi, sender, sender_counter, tx_power and contains collected RSSI sample data. The rssi and lqi fields carry the measured characteristics of the received beacon. The receiver_counter field is used to ensure that all data were downloaded. The sender, sender_counter and tx_level fields are the beacon parameters received as parameters of BeaconMsg. TransferFinishedMsg is sent by the receiver mote to the host computer. It notifies the experiment control script that the transfer was finished. It has a beacons_number field which carries the number of RSSI samples sent.

Moreover, 3 types of messages are used to allow debugging and reliability of the WSN application in situations where some mote is malfunctioning or some radio messages are lost. They are: ResetMsg, ConfirmationMsg and LogMsg. ResetMsg is broadcasted from the host computer to all the motes to put them in a default state at the start of each trial to provide mote synchronization. ConfirmationMsg(confirmation, from) allows to implement reliable message exchange. The confirmation field holds 0 or 1, and allows to discard duplicated messages received due to message retransmission. The from field holds the network address of the mote who sent the confirmation. LogMsg(text, value) allows to log the value of any integer variable with 12 character description text for debugging purposes.
Depending on the experiment stage, each message should be properly processed. So, the application logic is translated into a state machine, where each step of the experiment represents a separate state. There are several state machines running concurrently: one state machine controlling the experiment control script, and one state machine for each experiment mote. The state machine for the experiment control script behavior is illustrated in Figure 4.3, and consists of the following states:

**StateStart** allows users to choose an experiment scenario and establish connection to a serial forwarder software to communicate with the gateway mote. If the scenario file is loaded and connection to a WSN mote is established, **StateDeploy** is started.

**StateDeploy** keeps track of the current deployment, and increments the current deployment counter. If the current deployment counter is equal to the number of deployments in an experiment scenario, all deployments were performed and **StateFinish** is started. Otherwise, the coordinates of each mote are displayed and the script waits while the user places the motes as described and presses the button. Then, **StateTrial** is started.

**StateTrial** tracks the parameters of the trial. Each trial is specified by the number of beacons to be sent, the transmission power level and the delay between the beacons. The current trial counter is incremented, and if it is equal to the number of trials in a current deployment, the state is changed back to **StateDeploy**. Otherwise, **StateSender** is started.

**StateSender** chooses one mote to play the sender role. The mote counter is incremented, and if it is equal to the number of motes in the current deployment, all motes were given the sender role for the given trial, as the state is changed to **StateTrial**. Otherwise, measurements are started by the **StateReset** state.

**StateReset** sends the **ResetMsg** to each mote to provide synchronization between the motes. When all motes are reset, **StatePrepare** is started.

**StatePrepare** prepares all receivers for beacon reception. It sends **BeaconStartMsg** to every mote, except the sender mote allowing them to be prepared for the reception of beacon messages. When all receivers are ready, **StateMeasure** is started.

**StateMeasure** allows the sender to start sending beacons. It sends **BeaconStartMsg** to the sender mote and waits for the notification that all beacons were sent in form of **BeaconFinishedMsg** messages. This notification is expected during the particular time for recovery in case of sender failure. If no confirmation is
not received, next mote in a scenario is tried for the role of the sender. When BeaconFinishedMsg is received or the measurement task timeout is expired, StateTransfer is started.

**StateTransfer** enables the receiver motes to transfer collected measurements to the host machine. It sends TransferStartMsg to each receiver mote. On reception of an RSSI sample via TransferMsg, data are logged for further analysis. The end of transfer is signaled by the reception of TransferFinishedMsg. The transfer task timeout is set to be able to continue experiment execution in case of receiver mote failure. After all data are transferred, the state is changed to StateSender to give another mote the opportunity to send beacons.

**StateFinish** finishes the experiments, and saves collected data.

![State machine diagram](image)

Figure 4.3: State machine diagram for the experiment control script software behavior

The state machine for the experiment mote firmware behavior is illustrated in Figure 4.4, and consists of the following states:

**StateWait** enables each mote to wait for the trial start. When the BeaconStartMsg message from the host computer is received, the mote checks the sender field of the message. If the value of this field is equal to the mote’s address, the mote sets the trial parameters using the values from beacons_number, tx_power and delay message fields, and changes the state to StateBeacon. Otherwise, it changes the state to StateSample.
StateBeacon broadcasts beacons to the receiver motes. It sets the timer to fire with appropriate frequency, according to the trial parameters. When the timer is expired, the mote sets the transmission power level to a proper value and sends BeaconMsg to the receiver motes. The StateBeacon state keeps track of the number of beacons sent. When this number becomes equal to the number of beacons in the trial, the BeaconFinishedMsg message is sent to the host machine to inform the host machine that measurements were finished, then changes the state to StateWait. If ResetMsg is received during the operation, the measurement is aborted, and the state is changed to StateWait.

StateSample measures RSSI values when beacons from the sender mote are received. When BeaconMsg from the sender mote is received, the RSSI value is measured and stored into memory. When TransferStartMsg from the host computer is received, the state is changed to StateTransfer. If ResetMsg is received during sampling, the operation is aborted and the state is changed to StateWait.

StateTransfer transfers RSSI samples collected by the receiver mote to the host computer. It loads previously stored measurements, and sends them to the host machine sending TransferMsg. When all samples are transferred, it sends TransferFinishedMsg to the host machine. If ResetMsg is received during the transmission, the transfer is aborted, and the state is changed to StateWait.

![State machine diagram for the experiment mote firmware behavior](image-url)

Figure 4.4: State machine diagram for the experiment mote firmware behavior
4.2.4 Architecture and implementation of the experiment control script

The experiment control script is implemented in python and consists of several modules which can be semantically grouped into 3 layers. The high layer consists of two modules: Application and GUI (Graphical User Interface), which allows users to interact with the application. The Application module starts the script and initializes the other modules. The GUI module displays a user interface window, and allows users to control the application. The user interface is implemented using the wxPython library.

The middle layer of the script implements the experiment design logic and consists of 3 modules: Behavior, Data and Scenario. The Behavior module is responsible for appropriate processing of the WSN messages depending on the current experiment stage. The Scenario module deals with experiment scenario files. The Data module is responsible for saving the experiment result data for further analysis. The experiment results are saved in a CSV (Comma-Separated-Value) format convenient for further analysis.

The lower level of the script implements the services and utility functions used by the upper levels. It consists of 3 modules: WSN (Wireless Sensor Network), MVC (Model-View-Controller) and Statemachine. The WSN module allows the application to send and receive messages to and from the WSN motes. It is implemented using Twisted library, implementing the proper processing of data received from the serial forwarder application via TCP/IP socket. The binary data are grouped accordingly to the TinyOS message format. When a complete package is received, the application is notified. The Statemachine module is a generic state machine engine. It allows to implement the Behavior module as a set of separate classes, where each class is responsible for the single stage of the experiment. The MVC module allows to eliminate dependencies between other modules of the program, allowing all of them to be changed independently. The functionality of the application is represented as an exchange of events. The MVC module provides a mechanism which allows any module of the application to post events or subscribe to be notified when an event occurs. For example, the MVC module is used for implementing the state machine engine. The Statemachine module is keeping the list of the application states. When a specific state becomes current, it is subscribed to the application events to implement the script behavior. When the state stops being current, it is unsubscribed from the event notification.
4.2.5 Architecture and implementation of the experiment mote firmware

The experiment mote firmware consists of the following modules: Experiment, Behavior, Transport, Data and Logger. Like the main module of the application, the Experiment module provides the initialization of firmware parameters and implements the application specific functions, such as measuring RSSI, setting transmission power, saving and restoring collected data into memory before they are transferred to the host computer. The Behavior module implements the program logic, as defined by the experiment design. It contains 4 components: StateWaitP, StateBeaconP, StateSampleP, and StateTransferP. Each component implements 4 interfaces: StateMachine, StateEvent, AppEvent and AppFunction. Interfaces StateMachine and StateEvent allow a component to change the current state of the behavior and respond to state transition events. Interfaces AppEvent and AppFunction allow the component to perform application-related functions (for example, save/load last measured RSSI values to/from memory) and respond to application-related events (for example, to start sending beacon or to start collected data transfer). This approach is proven to facilitate the debugging of the firmware and allows to change the application behavior in order to support different experiments in a flexible manner. A state machine implementation in TinyOS was published in [67]. The Transport module detects situations where a radio message was lost, and resends data, providing reliable communication. The Logger transmits trace information to facilitate firmware debugging.

4.2.6 Communication protocol

During running pretests, we have observed that several messages sent by the motes are getting lost. No component providing reliable information transport was present in the version of TinyOS we were using. In this context, a communication protocol capable of detecting data loss and retransmitting lost messages was implemented. This communication protocol is a variation of simple Automatic Repeat Request protocol described in [69]. Both experiment control script and mote firmware maintain two sequence numbers. One number keeps the track of the next outgoing messages to be transmitted. Another number holds the sequence number of the next message expected. Sequence numbers of 1-bit (0 or 1) are sufficient in the given case, because the sender will not send the next message unless the acknowledgment for the previous message is received. At each moment, the receiver expects a particular sequence number. Any arriving frame containing the wrong sequence number is rejected as a duplicate. When a frame containing the correct sequence number arrives, it is accepted and passed to
the network layer. Then the expected sequence number is incremented modulo 2 (i.e., 0 becomes 1, and 1 becomes 0).

4.3 Experiment results

This section describes experiments that were run, and analyzes data that were collected in terms of RSSI measurements. More specifically, data from the first experiment provide descriptive statistics information about transmission ranges for different transmission power levels. The second experiment studies the variability in motes behavior. The third experiment studies the impact of the environment.

4.3.1 Experiment 1: Transmission ranges

The goal of the first experiment is to understand the transmission ranges and RSSI levels for different transmission power levels when motes are placed at different distances one from each other. In this case, what is the maximal range for each power level? What is the reliable reception range? Is there some range where RSSI values have smaller variation to serve, so localization will be more precise? Is it possible to use weak signals to improve precision of the localization? Is the picture at a highest transmission power level different from a picture at the lowest transmission power level?

Eight motes are placed at the distances of 0, 0.5, 1, 2, 3, 4, 8, 16 meters. Each mote acts as a sender, sending 30 beacons with transmission power levels of 1, 2, 3, 4, 5, 9, 10, 11, 19, 20, 21, 29, 30, 31. The means of the RSSI measures for selected values of transmission power levels are presented in Table 4.1.

For graphical comparison of RSSI values for each distance, boxplot diagrams are used. A boxplot is a way to graphically represent groups of data through five-number summaries: the smallest observation, the lower quartile (Q1), the median, the upper quartile (Q2), and the largest observation. A box in the middle of each boxplot depicts the range between the lower and the upper quartiles. A thick solid line across the box locates the median. To identify outliers, i.e. extreme values in the tails of the distribution, four quantities (called fences) are used: the lower inner fence (Q1−1.5*IQ, where IQ = Q2 − Q1), the upper inner fence (Q2 + 1.5*IQ), the lower outer fence (Q1 − 3*IQ), and the upper outer fence (Q2 + 3*IQ). A point beyond an inner fence on either side is considered a mild outlier. A point beyond an outer fence is considered
### Table 4.1: Variation of RSSI as a function of the distance.

<table>
<thead>
<tr>
<th>Distance (m)</th>
<th>TX=2</th>
<th>TX=3</th>
<th>TX=4</th>
<th>TX=10</th>
<th>TX=20</th>
<th>TX=30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>210.5926</td>
<td>227.8421</td>
<td>235.1121</td>
<td>239.9917</td>
<td>249.3750</td>
<td>177.6417</td>
</tr>
<tr>
<td>1</td>
<td>NA</td>
<td>222.4956</td>
<td>228.6844</td>
<td>236.1176</td>
<td>244.5375</td>
<td>246.3292</td>
</tr>
<tr>
<td>1.5</td>
<td>NA</td>
<td>227.5254</td>
<td>235.0167</td>
<td>241.0000</td>
<td>248.4167</td>
<td>253.9667</td>
</tr>
<tr>
<td>2</td>
<td>NA</td>
<td>220.8982</td>
<td>227.8282</td>
<td>234.5943</td>
<td>243.0500</td>
<td>245.7039</td>
</tr>
<tr>
<td>2.5</td>
<td>NA</td>
<td>215.8727</td>
<td>221.9655</td>
<td>229.8833</td>
<td>239.4833</td>
<td>243.1667</td>
</tr>
<tr>
<td>3</td>
<td>NA</td>
<td>224.4310</td>
<td>223.2091</td>
<td>230.3390</td>
<td>238.9565</td>
<td>242.2414</td>
</tr>
<tr>
<td>3.5</td>
<td>NA</td>
<td>220.9623</td>
<td>227.8500</td>
<td>234.9831</td>
<td>242.2069</td>
<td>245.5000</td>
</tr>
<tr>
<td>4</td>
<td>NA</td>
<td>217.9474</td>
<td>219.9626</td>
<td>225.8273</td>
<td>236.7611</td>
<td>240.0855</td>
</tr>
<tr>
<td>5</td>
<td>NA</td>
<td>217.7407</td>
<td>224.0000</td>
<td>231.1754</td>
<td>239.4167</td>
<td>244.7000</td>
</tr>
<tr>
<td>6</td>
<td>NA</td>
<td>209.6216</td>
<td>215.4314</td>
<td>223.0000</td>
<td>231.8833</td>
<td>236.4915</td>
</tr>
<tr>
<td>7</td>
<td>NA</td>
<td>213.4906</td>
<td>220.2549</td>
<td>227.3818</td>
<td>236.3559</td>
<td>239.5167</td>
</tr>
<tr>
<td>7.5</td>
<td>NA</td>
<td>212.6923</td>
<td>219.5789</td>
<td>226.8545</td>
<td>235.0345</td>
<td>238.8571</td>
</tr>
<tr>
<td>8</td>
<td>NA</td>
<td>211.5870</td>
<td>218.0400</td>
<td>224.5385</td>
<td>226.2522</td>
<td>229.0087</td>
</tr>
<tr>
<td>12</td>
<td>NA</td>
<td>212.0435</td>
<td>219.0179</td>
<td>226.1509</td>
<td>231.7368</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>NA</td>
<td>210.9643</td>
<td>217.9293</td>
<td>223.4643</td>
<td>231.5000</td>
<td>236.5000</td>
</tr>
<tr>
<td>14</td>
<td>NA</td>
<td>210.6087</td>
<td>217.4444</td>
<td>226.6000</td>
<td>231.1525</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>NA</td>
<td>210.0526</td>
<td>216.1489</td>
<td>222.9107</td>
<td>232.0508</td>
<td>236.0167</td>
</tr>
<tr>
<td>15.5</td>
<td>NA</td>
<td>209.1000</td>
<td>213.0000</td>
<td>220.4909</td>
<td>229.6379</td>
<td>233.2143</td>
</tr>
<tr>
<td>16</td>
<td>NA</td>
<td>NA</td>
<td>212.5536</td>
<td>223.6364</td>
<td>225.2881</td>
<td></td>
</tr>
</tbody>
</table>

Chapter 4. Implementation procedures, experiments and results
Chapter 4. Implementation procedures, experiments and results

an extreme outlier. Extreme outliers are drawn as small circles. Figure 4.5 compares a boxplot with a probability density function.

![Boxplot representation of the probability density function (pdf) of a normal population](image)

Figure 4.5: Boxplot representation of the probability density function (pdf) of a normal population

Figure 4.6 displays RSSI measured at transmission power levels 3, 4, 5. No motes were able to receive any beacons when the transmission power level was 1. At transmission power level 2, only the mote at distance 0.5 meter was able to receive beacons from the others. There is significant difference in RSSI levels measured at transmission power 3 in comparison with transmission power 4, whereas the difference in RSSI levels measured at transmission power 5 is insignificant. Figure 4.7 displays RSSI measured at transmission power levels 9, 10, 11. We realize that RSSI levels measured at transmission power levels 9, 10, 11 are similar to each other and higher than RSSI levels measured at transmission power levels 3, 4, 5. In the same vein, Figure 4.8 displays RSSI measured at transmission power levels 19, 20, 21. We realize that RSSI levels measured at transmission power 19, 20, 21 are similar to each other and higher than RSSI levels measured at transmission power levels 9, 10, 11. Finally, Figure 4.9 displays RSSI measured at transmission power levels 29, 30, 31. In this case, RSSI levels are similar to each other and higher than RSSI levels measured at transmission power levels 19, 20, 21.
Figure 4.6: RSSI vs Distance, for TX=3,4,5
Figure 4.7: RSSI vs Distance, TX=9,10,11
Figure 4.8: RSSI vs Distance, TX=19,20,21
Figure 4.9: RSSI vs Distance, TX=29,30,31
We can make several preliminary conclusions. First, all measured RSSI values are between 200 and 255 dBm, regardless the range of the RSSI register in CC2400 chip - which could provide values from 0 to 255 dBm. At low transmission power, the beacons are not received, as we see from Table 4.1. This means that the location information provided by the RSSI signal has low precision, i.e. two motes measuring the same RSSI level could be at different distances from the sending mote, which was unexpected. Second, increasing the transmission power level of the signal increases the measured RSSI level, showing that the hardware works as expected. However, all the measured results lie in a small range, as the difference in RSSI levels due to higher transmission power may be smaller than the fluctuations in RSSI measurements. Different transmission power levels provide different fluctuations in RSSI signals. Low power signal provides more precision at close distance, whereas high power signal provides less precision at close distance, where close distances are the distances below 5 meters. It does not look like any particular power level allows to have significantly better precision.

4.3.2 Experiment 2: Variability in motes behavior

The goal of the second experiment is to determine if all the motes can be considered equivalent in their abilities to measure RSSI levels. For this purpose, such an experiment was containing 4 deployments. Each deployment was using 9 motes with address equal to 9, 1, 2, 3, 4, 5, 6, 7, 8. Mote 9 was used as a reference mote. In each deployment, motes 1, 2, 3, 4, 5, 6, 7, 8 were positioned at the same distance from mote 9. This distance was equal to 1, 3, 7, 12 meters for each deployment. During each deployment, reference mote 9 was sending 15 beacons with transmission power level equal to 3, 5, 10, 20, 30 respectively. The mean values for RSSI measures for all devices at different transmission power levels are shown in Figures 4.10 to 4.14.

To verify if the differences in mean RSSI values can be attributed to the differences in the motes behavior, the analysis of variance (ANOVA) is performed. The analysis of this experiment is based on randomized block design [34]. A randomized block design allows to study the variation in the means of dependent variable called score (in our case, mean of RSSI values) in relation to two independent variables. The first independent variation is called treatment (parameter A), and represents the factor of the main interest in the studied phenomena (in the given experiment, it is the distance). The value of the jth distance level is denoted as $a_j$ (In our case, $a_1 = 1$, $a_2 = 3$, $a_3 = 7$, $a_4 = 12$). The second independent variable (parameter $BL$) represents the differences among the experiment units which may make a contribution to error variation and thereby mask or obscure the treatment effects (in the given experiment, it is the variation in motes behavior). The variation in the dependent variable attributable to such sources is called nuisance.
Figure 4.10: RSSI vs Distance for all motes for TX=3
Figure 4.11: RSSI vs Distance for all motes for TX=5
Figure 4.12: RSSI vs Distance for all motes for TX=10
Figure 4.13: RSSI vs Distance for all motes for TX=20
Figure 4.14: RSSI vs Distance for all motes for TX=30
variation. To isolate nuisance variation, a blocking procedure is used. The blocking procedure involves forming \( n \) blocks of \( m \) homogeneous experimental units, where \( m \) is the number of treatment levels and \( n \) is the number of the levels of nuisance variable. In the given experiment, \( n \) is equal to the number of motes and equal to 8, and \( m \) is the number of different distance levels and equal to 4. The value of the \( i \)th level of nuisance variable is denoted \( s_i \). In our case, \( s_i \) corresponds to the different mote addresses, \( s_1 = 1, s_2 = 2, s_3 = 3, s_4 = 4, s_5 = 5, s_6 = 6, s_7 = 7, s_8 = 8 \). This design is denoted by letters RB-\( m \), as experiment 2 is called an RB-4 experiment.

In a randomized block design, a score \( Y_{ij} \) is a mean value that reflects the effects of distance level \( i \), for the mote \( j \), and all other sources of variation that affect \( Y_{ij} \). These latter sources of variation are collectively referred to as residual effects or error effects. The expectation \( Y_{ij} \) is expressed more formally as follows:

\[
Y_{ij} = \mu + \alpha_j + \pi_i + \epsilon_{ij} \\
(i = 1, n, n = 8; j = 1, m, m = 4)
\]

where

\( Y_{ij} \) is the mean RSSI value for distance level \( j \) and mote \( i \);

\( \mu \) is the grand mean of RSSI values \( \mu_1, \mu_2, \ldots, \mu_{nm} \), where \( \mu_{nm} \) is the RSSI value measured by mote \( n \) at distance \( m \), \( n = 8 \), and \( m = 4 \);

\( \alpha_j \) is the treatment effect and is equal to \( \mu_j - \mu \), the deviation of the grand mean from the mean RSSI values for distance level \( j \). The \( j \)th treatment effect is a constant for mean RSSI values in distance \( a_j \), and is subject to the restriction \( \sum_{j=1}^{m} \alpha_j = 0 \);

\( \pi_i \) is the block effect for particular mote \( i \), and is equal to \( \mu_i - \mu \), the deviation of the grand mean from the RSSI mean value for mote \( i \). The block effect is a normally and independently distributed random variable with mean 0 and standard deviation \( \sigma^2_\pi \) (denoted as \( \text{NID}(0, \sigma^2_\pi) \));

\( \epsilon_{ij} \) is the error associated with \( Y_{ij} \), and is equal to \( Y_{ij} - \mu_j - \mu_i + \mu \). The error effect is a random variable that is \( \text{NID}(0, \sigma^2_\epsilon) \) and independent of \( \pi_i \).

The values of parameters \( \mu, \alpha_j, \pi_i \) and \( \epsilon_{ij} \) in model (4.2) are unknown. But, they can be estimated from sample data as follows:

\[
Y_{ij} = \bar{Y}_\cdot + (\bar{Y}_j - \bar{Y}_\cdot) \\
\]

\( \bar{Y}_\cdot \) Grand mean

\( \bar{Y}_j - \bar{Y}_\cdot \) Effect of the distance
Equation (4.3) can be rearranged as follows:

\[ Y_{ij} - \bar{Y}_\cdot = (\bar{Y}_{.j} - \bar{Y}_\cdot) + (\bar{Y}_{i.} - \bar{Y}_\cdot) + (\bar{Y}_{ij} - \bar{Y}_{.j} - \bar{Y}_{i.} + \bar{Y}_\cdot) \]  

(4.4)

Next, we square both sides of (4.4) as follows:

\[ (Y_{ij} - \bar{Y}_\cdot)^2 = [(\bar{Y}_{.j} - \bar{Y}_\cdot) + (\bar{Y}_{i.} - \bar{Y}_\cdot) + (\bar{Y}_{ij} - \bar{Y}_{.j} - \bar{Y}_{i.} + \bar{Y}_\cdot)]^2 \]  

(4.5)

From (4.5), the total sum of squares (\(SSTO\)) could be partitioned into three parts: the sum of squares due to the distance effect (\(SSA\)), the sum of squares due to motes difference (\(SSBL\)), and the sum of squares due to residual errors (\(SSRES\)). This sum can be expressed as follows:

\[
SSTO = \sum_{j=1}^{m} \sum_{i=1}^{n} (Y_{ij} - \bar{Y}_\cdot)^2 = n \sum_{j=1}^{m} (\bar{Y}_{.j} - \bar{Y}_\cdot)^2 + p \sum_{i=1}^{n} (\bar{Y}_{i.} - \bar{Y}_\cdot)^2 + \sum_{j=1}^{m} \sum_{i=1}^{n} (\bar{Y}_{ij} - \bar{Y}_{.j} - \bar{Y}_{i.} + \bar{Y}_\cdot)^2
\]

\[
(4.6)
\]

The randomized block design operates by calculating the total sum of squares \(SSTO\) and partitioning this sum into three parts: \(SSA\), \(SSBL\) and \(SSRES\).

The mean squares (MS) are obtained by dividing each sum of squares by its degree of freedom [34]. More precisely, we have:

\[
MSTO = \frac{SSTO}{nm - 1}
\]

\[
MSA = \frac{SSA}{m - 1}
\]

\[
MSBL = \frac{SSBL}{n - 1}
\]

\[
MSRES = \frac{SSRES}{(n - 1)(m - 1)}
\]

The goal of the analysis is to check two hypotheses \(H_0\) and \(H_1\), first for the distance effect, then for the motes effect. The statistical hypotheses for the distance effect are:
Chapter 4. Implementation procedures, experiments and results

67

\[ H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4 \quad \text{or} \quad H_0 : \forall j, \alpha_j = 0 \quad \text{for} \quad j \in \overline{1, 4} \]

\[ H_1 : \mu_j \neq \mu_{j'} \quad \text{for} \quad j \neq j' \quad \text{where} \quad j, j' \in \overline{1, 4} \quad \text{or} \quad H_1 : \exists j, \alpha_j \neq 0 \quad \text{for} \quad j \in \overline{1, 4} \]

(4.7)

A test of null hypothesis is given by \( FA = \frac{MSA}{MSRES} \) [34]. If the null hypothesis is true, the \( F \) statistic is distributed as the \( F \) distribution with \( m - 1 \) and \( (n - 1)(m - 1) \) degrees of freedom. An \( F \) statistic and \( F \) distribution are used to test a hypothesis about two population variances, rather than a single variance. An \( F \) random variable is defined as the ratio of two independent chi-square variables, each divided by its degrees of freedom: \( F = \frac{\chi^2(d_1) / d_1}{\chi^2(d_2) / d_2} \). The distribution \( F \) is used to determine probability \( p \) of observing the \( F \) statistic as large as or larger than the statistic obtained. According to convention, an \( F \) statistic that falls in the upper 5% of the sampling distribution of \( F \) is considered to be sufficient evidence for rejecting the null hypotheses.

Moreover, the statistical hypotheses for the mote differences are:

\[ H_0 : \sigma^2_\pi = 0 \]

\[ H_1 : \sigma^2_\pi \neq 0 \]

(4.8)

A test of null hypothesis for motes differences is given by \( FBL = \frac{MSBL}{MSRES} \) [34]. If the null hypothesis is true, this \( F \) statistic is distributed as \( F \) with \( n - 1 \) and \( (n - 1)(m - 1) \) degrees of freedom.

The analysis for the situation where motes 1-8 were receiving beacons from the reference mote, and the situation where motes 1-8 were sending beacons to the reference mote were performed separately. Results for the situation when motes 1-8 was receiving beacons from reference mote 9 are presented in Table 4.2, whereas analysis results for the situation where motes 1-8 were sending beacons from reference mote 9 are presented in Table 4.3. These tables specify the degree of freedom \( (df) \), the sum of squares \( (SS) \), the mean square \( (MS) \), as well as the \( F \) statistic and the probability \( (p) \) for each variable.

For the situation where motes 1-8 were receiving beacons, we obtained: \( FBL = 0.55, p = 0.79 > 0.05 \). It means that the differences in RSSI values due to the motes difference are non significant. Similarly, we obtained: \( FA = 1.56, p = 0.23 \). As a result, the variation due to the distance is not significant. For the situation where
motes 1-8 was sending beacons, we obtained: $F_{BL} = 0.96$, $p = 0.48 > 0.05$. It means that the differences in RSSI values due to the difference in motes are non significant. Similarly, we obtained: $F_A = 1.60$, $p = 0.22$. So, the variation due to the distance is not significant.

It may be noted that, in both cases, the effect of the distance is not significant. We attribute this to the fact that our statistics were taking into account the performance of the motes at all transmission power levels together, as the difference at RSSI recorded for different transmission power masks the effect of the changing distance. This was done to focus on block effects, this analysis allows to take into account possible interaction between two factors - transmission power levels and the performance of each separate mote.

The results of experiment 2 allow to conclude that the motes are manufactured with enough precision, so that no calibration is needed to compensate difference in the motes behavior.

### 4.3.3 Experiment 3: Impact of the environment

The goal of the third experiment is to evaluate the environment impact on RSSI measurements. For this purpose, 9 motes were deployed in two different environments. The first environment is a large classroom filled with chairs and tables. The second environment is a narrow and long underground tunnel without any furniture. Each deployment was using 9 motes with addresses equal to 9, 1, 2, 3, 4, 5, 6, 7, 8. Mote 9 was used as a reference mote. In each deployment, motes 1, 2, 3, 4, 5, 6, 7, 8 were

### Table 4.2: Variation of RSSI due to the motes difference (reception)

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
<td>7</td>
<td>484.29</td>
<td>69.18</td>
<td>0.55</td>
<td>0.79</td>
</tr>
<tr>
<td>Distance</td>
<td>3</td>
<td>585.83</td>
<td>195.28</td>
<td>1.556</td>
<td>0.23</td>
</tr>
<tr>
<td>Residuals</td>
<td>21</td>
<td>2635.44</td>
<td>125.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 4.3: Variation of RSSI due to the motes difference (transmission)

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>7</td>
<td>2535.83</td>
<td>362.26</td>
<td>0.96</td>
<td>0.48</td>
</tr>
<tr>
<td>Distance</td>
<td>3</td>
<td>1814.2</td>
<td>604.7</td>
<td>1.5955</td>
<td>0.22</td>
</tr>
<tr>
<td>Residuals</td>
<td>21</td>
<td>7959.3</td>
<td>379.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
positioned at the same distance from mote 9. This distance was equal to 1, 2, 3, 5, 6, 8, 10 meters. During each deployment, reference mote 9 was sending 15 beacons with transmission power 5, 10, 20, 30, then it was receiving 15 beacons with transmission power 5, 10, 20, 30 from each other mote. The mean values of RSSI of all motes at different transmission power levels in different environments are shown in Figures 4.15 to 4.18.

![Figure 4.15: RSSI vs Distance for 2 environments with TX=5](image)

To verify if the differences in RSSI means can be attributed to the differences in environment, the analysis of variance is performed. Experiment 3 has a randomized block factorial denoted $(RBF-mq)$ design [34]. In this design, the differences in the mean RSSI values $(Y_{ijk})$ are attributed to three factors: environment (factor $A$), distance (factor $B$), and blocking variable $(BL)$. The environment variable has $p = 2$ levels, denoted $a_j$, where $a_1 = 1$ corresponds to the classroom, and $a_2 = 2$ corresponds to the tunnel. The distance variable has $q = 7$ levels, denoted $b_k$, where $b_1 = 1, b_2 = 2, b_3 = 3, b_4 = 5, b_5 = 6, b_6 = 8, b_7 = 10$ meters. The blocking variable in this case
Figure 4.16: RSSI vs Distance for 2 environments with TX=10
Figure 4.17: RSSI vs Distance for 2 environments with TX=20
Figure 4.18: RSSI vs Distance for 2 environments with TX=30
represents different mote addresses and has $n = 8$ levels denoted $s_i$, $i \in \Gamma, \lambda$. This design is specified by the following model:

\[
Y_{ijk} = \mu + \alpha_j + \beta_k + (\alpha\beta)_{jk} + \pi_i + (\alpha\beta\pi)_{jki} + \epsilon_{ijk},
\]

\[i = 1, n; j = 1, p; p = 2; k = 1, q; q = 7\] (4.9)

where

$Y_{ijk}$ is the mean RSSI value for mote $i$, environment $a_j$, and distance $b_k$;

$\mu$ is the grand mean of the RSSI values;

$\alpha_j$ is the effect of the environment (parameter $A$) and is subject to the restriction $\sum_{j=1}^{p} \alpha_j = 0$, $p = 2$;

$\beta_k$ is the effect of the distance (parameter $B$) and is subject to the restriction $\sum_{k=1}^{q} \beta_k = 0$, $q = 7$;

$(\alpha\beta)_{jk}$ is the joint effect of environment level $j$ and distance level $k$, and is subject to restrictions $\sum_{j=1}^{p} (\alpha\beta)_{jk} = 0$ and $\sum_{k=1}^{q} (\alpha\beta)_{jk} = 0$, $p = 2$ and $q = 7$;

$\pi_i$ is the effect attributed to the differences in mote $i$, and is $NID(0, \sigma^2_{\pi})$.

$(\alpha\beta\pi)_{jki}$ is the joint effect of environment $a_j$ and distance $b_k$ on mote $i$; $(\alpha\beta\pi)_{jki}$ is $NID(0, \sigma^2_{\alpha\beta\pi})$ and independent of $\pi_i$.

$\epsilon_{ijk}$ is the error effect that is $NID(0, \sigma^2_{\epsilon})$ and independent of $\pi_i$. In this design, $\epsilon_{ijk}$ cannot be estimated separately from $(\alpha\beta\pi)_{jki}$.

In this case, the total sum of squares ($SSTO$) can be partitioned into five parts as follows:

$$SSTO = SSA \quad \text{(Environment effect)}$$
$$+ SSB \quad \text{(Distance effect)}$$
$$+ SSAB \quad \text{(Interaction environment-distance effect)}$$
$$+ SSBL \quad \text{(Motes effect)}$$
$$+ SSRES \quad \text{(Errors)}$$
The mean squares (MS) are obtained by dividing each sum of squares by its degree of freedom as follows:

\[
\begin{align*}
MSTO &= \frac{SSTO}{nmq - 1} \\
MSA &= \frac{SSA}{m - 1} \\
MSB &= \frac{SSB}{q - 1} \\
MSBL &= \frac{SSBL}{n - 1} \\
MSAB &= \frac{SSAB}{(m - 1)(q - 1)} \\
MSRES &= \frac{SSRES}{(n - 1)(mq - 1)}
\end{align*}
\]

A test of null hypothesis is performed by calculating separate F statistics for each factor, their combination and block parameter as follows:

\[
\begin{align*}
FA &= \frac{MSA}{MSRES} \\
FB &= \frac{MSB}{MSRES} \\
FBL &= \frac{MSBL}{MSRES} \\
FAB &= \frac{MSAB}{MSRES}
\end{align*}
\]

Then, the probability \( p \) of obtaining the value of \( F \) that is equal or more extreme than observed is calculated. The null hypothesis is rejected if the probability value is less than or equal to a preselected level of significance. As in experiment 2, the confidence level \( \alpha = 0.05 \) is used, which is a traditional convention for ANOVA tests.

Results were analyzed separately for transmission and reception performance. Results analysis for the situation where motes 1-8 were receiving beacons from the reference mote 9 is presented in Table 4.4. We realize that:

- Differences in RSSI values due to different environments are significant \( (p = 0.003069 < 0.05) \);
Chapter 4. Implementation procedures, experiments and results

- Differences in RSSI values due to different distances are significant \((p = 1.240 \times 10^{-11} < 0.05)\);

- Differences in RSSI values due to combination of environments and distances are significant \((p = 0.0002173 < 0.05)\).

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>1</td>
<td>513.91</td>
<td>513.91</td>
<td>19.567</td>
<td>0.003069</td>
</tr>
<tr>
<td>Distance</td>
<td>6</td>
<td>2897.22</td>
<td>482.87</td>
<td>22.392</td>
<td>(1.240 \times 10^{-11})</td>
</tr>
<tr>
<td>Environment*Distance</td>
<td>6</td>
<td>833.73</td>
<td>138.95</td>
<td>5.6777</td>
<td>0.0002173</td>
</tr>
</tbody>
</table>

Table 4.4: Variation of RSSI due to the environment (reception)

Results analysis for the situation where motes 1-8 were sending beacons to the reference mote 9 is presented in Table 4.5. From these results, we realize that:

- Differences in RSSI values due to different environments are non-significant \((p = 0.933 > 0.05)\);

- Differences in RSSI values due to different distance are significant \((p = 0.00973 < 0.05)\);

- Differences in RSSI values due to combination of environments and distances are significant \((p = 0.0001056)\).

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment</td>
<td>1</td>
<td>1.30</td>
<td>1.30</td>
<td>0.0076</td>
<td>0.933</td>
</tr>
<tr>
<td>Distance</td>
<td>6</td>
<td>2663.7</td>
<td>444.0</td>
<td>3.282</td>
<td>0.00973</td>
</tr>
<tr>
<td>Environment*Distance</td>
<td>6</td>
<td>4132.5</td>
<td>688.8</td>
<td>4.6429</td>
<td>0.001056</td>
</tr>
</tbody>
</table>

Table 4.5: Variation of RSSI due to the environment (transmission)

The results of experiment 3 allow to conclude that motes deployed in different environments will need separate calibration for each environment. Consequently, experiment 3 can be used to validate results from experiment 2. To do this, measurements from classroom and tunnel were analyzed separately to verify if all differences in RSSI measurements can be attributed to the differences between the motes. The results of this analysis are presented in Tables 4.6 to 4.9.

In general, the variation in RSSI values due to the differences between the motes is non-significant. More specifically, we obtained:
• For reception in classroom: $p = 0.2180 > 0.05$ (non-significant);
• For transmission in classroom: $p = 0.4109 > 0.05$ (non-significant);
• For reception in tunnel: $p = 0.00267 < 0.05$ (significant);
• For transmission in tunnel: $p = 0.1642 > 0.05$ (non-significant);

This confirms the conclusion made when analyzing the results of experiment 2.

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>7</td>
<td>324.94</td>
<td>46.42</td>
<td>1.4237</td>
<td>0.2180</td>
</tr>
<tr>
<td>Distance</td>
<td>6</td>
<td>1033.25</td>
<td>172.21</td>
<td>5.3148</td>
<td>0.0003742</td>
</tr>
<tr>
<td>Residuals</td>
<td>42</td>
<td>1360.86</td>
<td>32.40</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.6: Differences between motes (classroom, reception)

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
<td>7</td>
<td>2004.8</td>
<td>286.4</td>
<td>1.0518</td>
<td>0.4109</td>
</tr>
<tr>
<td>Distance</td>
<td>6</td>
<td>4255.0</td>
<td>709.2</td>
<td>2.6046</td>
<td>0.03086</td>
</tr>
<tr>
<td>Residuals</td>
<td>42</td>
<td>11435.7</td>
<td>272.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7: Differences between motes (classroom, transmission)

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>7</td>
<td>365.12</td>
<td>52.16</td>
<td>3.82</td>
<td>0.00267</td>
</tr>
<tr>
<td>Distance</td>
<td>6</td>
<td>2697.69</td>
<td>449.62</td>
<td>32.972</td>
<td>$2.262 \times 10^{-14}$</td>
</tr>
<tr>
<td>Residuals</td>
<td>42</td>
<td>572.72</td>
<td>13.64</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.8: Differences between motes (tunnel, reception)
### Table 4.9: Differences between motes (tunnel, transmission)

<table>
<thead>
<tr>
<th>Source</th>
<th>df</th>
<th>SS</th>
<th>MS</th>
<th>F</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiver</td>
<td>7</td>
<td>126.451</td>
<td>18.064</td>
<td>1.5930</td>
<td>0.1642</td>
</tr>
<tr>
<td>Distance</td>
<td>6</td>
<td>2541.21</td>
<td>423.53</td>
<td>37.353</td>
<td>$2.648 \times 10^{-15}$</td>
</tr>
<tr>
<td>Residuals</td>
<td>42</td>
<td>476.22</td>
<td>11.34</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 5

Conclusion

After presenting the importance of localization in WSN applications, as well as the main methods and algorithms for localization, we have implemented a WSN application for RSSI properties evaluation. In this chapter, we will give some concluding remarks. More specifically, this chapter summarizes the work done and the results obtained, states the limitations, and outlines the directions for future work.

5.1 Summary

One of the main characteristics of WSN is that they are application dependent. In Chapter 2, several existing WSN applications are reviewed. Based on their interaction patterns, WSN applications are divided into four groups: event detection, periodic measurements, function approximation, and tracking. It is concluded that depending on application requirements, different services need to be implemented by WSN, and different hardware and software solutions are used. It is observed that the localization service is very important for many WSN applications, especially for tracking and function approximation applications. It is also noted that, in many situations, using existing localization solutions, like GPS, is not possible [37], [12]. Even if an application is deployed outdoors, like in [7], [3], GPS may be prohibitively expensive for the networks with a great number of nodes. Thus, localization needs to be implemented by WSN.

In Chapter 3, existing solutions for the node localization problem in WSN are reviewed. Three approaches to measure distances between the nodes (ranging) are dis-
cussed: Time of Arrival (ToA), Time Difference of Arrival (TDoA), Received Signal Strength Indicator (RSSI). ToA and TDoA methods are known to achieve good accuracy. But, they are expensive in terms of extra hardware which needs to be present at each network node. The RSSI ranging method is more accessible, as no additional hardware is needed. But, the measured distance is known to be less precise. Moreover, the approaches to determine the exact position of the node are divided into two groups: geometric positioning methods and probabilistic positioning methods. The geometric positioning methods use geometric relations to calculate the nodes positions. They are known to be quite precise, as they demand precise ranging measurements for correct positioning [77], [2]. The probabilistic methods try to reduce the effect of ranging errors, when less precise ranging methods are used, by using probability theory [79]. A theoretical bound for the multi-hop localization methods performance is presented. Several concrete examples of implemented WSN localization algorithms are discussed.

The experiment design, the implementation and the results are presented in Chapter 4, where a universal WSN application to measure RSSI is implemented. The application consists of two parts: an experiment control script which runs on a computer, and an experiment mote firmware which runs on each WSN node. The experiment control script sends experiment configuration parameters to each node, commands each node to start measurements accordingly to experiment scenarios, and collects measured data. The experiment mote firmware performs RSSI measurements.

Statistical analysis of variance (ANOVA) was performed to determine the factors affecting the RSSI measurements. Results analysis shows that measured RSSI values have significant variation. In situations where two wireless motes were located at the distance less than 10 meters from each other, the measured RSSI values were in the range between 200 and 250 dBm. However, it was noticed that RSSI values measured with the maximal transmission power at distances of about 1 meter can be small (in the range between 0 and 30 dBm), which may be used for proximity detection. Therefore, it was not be possible to directly evaluate the distance between two motes with a precision suitable for the range-based positioning algorithms using the RSSI measurements. Also, it was shown that:

- The relation between RSSI values and distances depends on the environment. As a result, any real localization system using the RSSI needs to be calibrated for each specific deployment;

- The used WSN motes are manufactured with enough precision, as the differences between devices are insignificant. As a result, different motes do not need to be calibrated for use in localization system;
• Even if the RSSI measurements have significant variation, the mean RSSI values correlate with the distances. As a result, RSSI information can be used for positioning. Expected ranging error would be in a order of 5-10 meters;

• Using different transmission power levels can provide additional information about the distances.

5.2 Limitations

Even if we reached the research objectives, we can mention several limitations related to this research. First, we only studied the properties of RSSI measurements for the purpose of ranging. We did not implement any particular positioning algorithm. Second, we observed small range and large variation in the measured RSSI values due to the limitations of the existing motes hardware, which would increase ranging errors. Also, in our experiments, all antennas had the same orientation to focus on the dependence between RSSI and distances, and it was assumed that the environment around the motes did not change during the experiments. It may not be the case for dynamic environments or long deployments. Finally, during data analysis, we did not analyze data about missing packets. It is probable that this information is correlated with the distance, and may be used to assist localization.

5.3 Future work

Future work should be oriented towards the following points. First, the positioning methods, which take into account the variability in RSSI measurements, need to be developed. Second, for using in geometrical positioning methods, further research may provide a way to filter noisy RSSI data to improve ranging precision. Also, when using probabilistic positioning methods, techniques to store, collect and update the RSSI database need to be developed. In the same vein, robust WSN architectures capable of aggregating and exchanging RSSI information in a distributed and energy-efficient manner need to be created. Another problem to solve is to find a way to update RSSI information in order to take into account the changes in the environment.

Also, it may be interesting to exploit application-specific information received from the sensors to improve localization precision. For example, imagine a WSN with moving nodes which use the anchor-based localization system. If there is a way to sense that
each given mobile node is very close to the given anchor, this information can be used as a feedback for the localization algorithm to improve precision. The proximity detection is often a simpler problem than ranging, and could be already implemented as a part of the application functionality.
Bibliography


Appendix A

Experiment mote firmware source code

This appendix presents the source code for experiment mote firmware.

The experiment mote firmware is implemented in nesC, programming language of TinyOS [22], a special language which is designed for sensor nodes programming. NesC is designed to facilitate writing programs which are driven by the interaction with an environment, have limited resources and should be reliable. To achieve this, nesC has several specific features.

First, nesC is designed to support split-phase operations. All operations which can cause some delay (for example, obtaining measurements from sensors or wireless communications or waiting for sensor measurements) are split into two functions: operation request (called command) and operation completion (called event). Event functions are callbacks which get called when a particular condition is fired.

Second, NesC has support for concurrency. Its concurrency model consists of two levels: interrupts and tasks. Interrupts related to hardware interrupts and can occur at any time. Tasks are delayed procedure calls that are executed within the context of the component, and run to completion. Interrupt can preempt tasks, whereas one task cannot preempt another task.

Third, the nesC programs have an explicit linking. The program is composed from reusable components. There are two types of components: modules and configurations. A module may contain variables and implement functions. A configuration explicitly connects components together, allowing one component to use the functions of another component. There are two types of functions: commands and events. Commands work as simple functions - one component calls a command when it wants another component
to perform some operations. Events work as callbacks - they are called by others to inform a component about a specific condition. The functions are made visible for other modules by using interfaces. Interfaces group logically related functions together. Each component of nesC declares the interfaces it provides to other components and the interfaces of other components it uses. Each component has its private data, providing data protection. All components in nesC program have local namespaces and cannot access variables from other modules. All memory of the program is allocated statically at compiling time.

A program written in nesC is processed by a code generator which combines the code of each component into the resulting program. The design of the language makes it possible to perform static analysis and optimization, providing the way to detect some potential errors (notably, errors related to concurrency problems) and to perform code optimization.

The main configuration of the firmware, ExperimentC, is implemented in Listing A.1 and shown in Figure A.1. Graphical notation of a diagram is traditionally used in other publications about TinyOS [22], [21]. It is called a wiring diagram. Semantically, it is equivalent of a UML component diagram. It shows the subdivision of a program into parts and dependencies between them. Boxes of the diagram represent the nesC modules, arrows represent interfaces. The arrows on the left side of the block are the interfaces used by the component. The arrows on the right side of the block are the interfaces provided by the component. Black arrows represent the interfaces which implement the commands. White arrows represent the events.

![Figure A.1: Main experiment mote firmware implementation wiring diagram](image-url)
The main module of the firmware *ExperimentM*, shown in Listing A.2, provides 3 interfaces: *StdControl*, *AppEvent*, and *AppFunctions*. Interface *StdControl* is an entry point of application. It is used by the *Main* component, provided by TinyOS, to initialize application and start execution. An interface *AppEvent*, shown in Listing A.3, contains all events which can change the application behavior. Notably, the events correspond to messages sent by the experiment control script as described it 4.2.3. The *ExperimentM* module translates “raw” events from hardware and communication protocols into “high-level” application events, and handles them to the *BehaviorC* component to be processed accordingly to the current state. An interface *AppFunction*, shown in Listing A.4 represents all application-specific features. More specifically, it contains functions which allow to send radio messages, blink leds for debugging purposes, set transmission power level, measure RSSI values and store them for further retrieval. To achieve this it uses functions from components *TimerC*, *LedsC*, *CC2420RadioC* provided by TinyOS to control mote’s hardware, and component *TransportC*, shown in Listing A.16, which provides reliable message transport with a retransmission. The communication protocol is implemented in the module *TransportM*, shown in Listing A.17

The application logic is programmed as a state machine. State machines and statecharts [24] are traditional techniques used for designing event-driven systems. They allow to formalize the behavior of the system in a form that can be easily documented, verified and implemented. State machines are a well established methods to define systems with complex behavior. There are two reasons for using state machines in WSN programming. First, because of the application requirements, a WSN node should perform different functions depending on the external conditions or the stage of deployment. In this scenario, each node should have different reactions on each type of event depending on what it is doing at the current moment. For example, if a node is doing configuration, and a sensing event occurred, the sensing event should be ignored because the parameters of the sensing are not set up properly. If a sensing event occurs in a measurement phase of deployment, the data from the sensors should be stored into memory. For a complex behavior, with many events and many states, checking necessary conditions directly inside the event handlers may become cumbersome and error prone. If the application’s requirements change, adding new functionality may demand to change many event handlers, making modification of the program difficult. Second, the split-phase model of execution demands the programmer to include additional checks in the code to make sure that components data are treated correctly in the commands and events. As nesC is based on C, on a level of a single module, state machine can be represented by having a switch operator inside of every event handler which executes the code appropriate for the current state. Unfortunately, this approach does not scale well. The introduction of new states demands changing of every event
handler, and considering that all module can only access its internal variables, modules
tend to grow long and have many shared variables, which is error-prone.

The proposed state machine implementation is using the special features of the
language for the advantage. The idea is to present each state as a separate component.
It allows to group the processing of events relevant to each phase of an application into
separate logical components. This simplifies changing the behavior of the program and
makes simpler to verify that the program behaves as specified. The code works using
several features specific to nesC. First, parameterized interfaces are used to express
each module representing particular state of the application. Parameterized interfaces
act as arrays of interfaces and provide ability to check at compile time that all modules
are implementing required functions. There is a way to assign each interface an unique
number to link each instance of interface to a specific implementation. Second, split
phase operations are used for delivering events from application to the current state for
processing. Third, tasks are used inside the state machine components, so compiler is
able to check for potential race conditions.

The firmware behavior, corresponding to the state machine displayed in Figure 4.4,
is specified in BehaviorC configuration, implemented in Listing A.9 and shown in Fig-
ure A.2. The behavior uses three types of modules. The first module, ExperimentM,
represents an application. It uses the BehaviorM module implemented in Listing A.10
to control behavior via Behavior interface declared in Listing A.8. The second module,
StateMachineM, shown in Listing A.7, is a state machine engine which tracks the cur-
rent state. It dispatches every application event received from the ExperimentM mod-
ule via AppEvent interface to the appropriate state’s AppEvent interface. ExperimentM
module allows BehaviorM module to change its current state, and to ask what is
the current state via StateMachine interface, shown in Listing A.6. When a state
transition occurs, the StateMachine module informs the current state component
and the next state component via StateEvent interface shown in Listing A.5. The
third group of modules are the modules StateWaitP, StateBeaconP, StateSampleP,
StateTransferP, shown in Listings A.14, A.11, A.12, and A.13, which implement the
specific behavior of the application in a given context.

The proper context is guaranteed by the StateMachine module. Each state re-
ceives application events via AppEvent interface, signaled by the StateMachine mod-
ule. Each state is able to initiate state transition using the StateMachine interface
of StateMachine module. Every state is informed when a state was changed via the
StateEvent interface of StateMachine module.

Modules LoggerC and LoggerM, shown in Listings A.29, A.30 implement the ability
Figure A.2: Mote behavior implementation wiring diagram
to send debugging information via a serial cable. Debug events are buffered in a circular buffer implemented in Listing A.27.

Listing A.1: ExperimentC.nc - Main wiring configuration of mote firmware

```c
configuration ExperimentC {
}
implementation {
  components Main
    , ExperimentM
    , TransportC
    , TimerC
    , BehaviorC
    , LedsC
    , GenericComm
    , CC2420RadioC
    ;
  Main.StdControl -> ExperimentM;
  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> CC2420RadioC;
  Main.StdControl -> GenericComm;
  ExperimentM.Leds -> LedsC.Leds;
  ExperimentM.RadioControl -> CC2420RadioC.CC2420Control;
  ExperimentM.Timer -> TimerC.Timer[unique("Timer")];
  ExperimentM.BlinkTimer -> TimerC.Timer[unique("Timer")];
  ExperimentM.Transport -> TransportC.Transport;
  ExperimentM.Behavior -> BehaviorC.Behavior;
  BehaviorC.AppEvent -> ExperimentM.AppEvent;
}
```

Listing A.2: ExperimentM.nc - Main application operations

```c
includes ResetMsg;
includes BeaconStartMsg;
includes BeaconMsg;
includes TransferStartMsg;
includes BeaconFinishedMsg;
includes TransferMsg;
includes TransferFinishedMsg;
includes ConfirmationMsg;

module ExperimentM{
  provides {
    interface StdControl;
    interface AppEvent;
    interface AppFunction;
  }
  uses {
    interface Leds;
    interface CC2420Control as RadioControl;
    interface Behavior;
    interface Transport;
    interface Timer;
    interface Timer as BlinkTimer;
  }
  implementation{
    enum {MAX_STORAGE_LEN=100};
    ```
struct TOS_Msg m_msg;
uint16_t m_counter;
uint8_t m_tx_power;

uint16_t m_storage_length;
Sample m_storage[MAX_STORAGE_LEN];
AppData m_data;

//======================================================================
// Interface StdControl.
//======================================================================
command result_t StdControl.init(){
    m_storage_length = 0;
call Leds.init();
return SUCCESS;
}

command result_t StdControl.start(){
    m_storage_length = 0;
call Behavior.start();
return SUCCESS;
}

command result_t StdControl.stop(){
return SUCCESS;
}

//=====================================================================
// Interface AppFunction.
//=====================================================================
command result_t AppFunction.sendBeaconMsg(
    uint16_t a_sender
    , uint16_t a_counter
    , uint8_t a_tx_power
){
    BeaconMsg_t *msg = (BeaconMsg_t*) m_msg.data;
    bool rv = FALSE;
    msg->sender = a_sender;
    msg->counter = a_counter;
    msg->tx_power = a_tx_power;
call Transport.sendBeaconMsg(&m_msg);
rv = SUCCESS;

return rv;
}

command result_t AppFunction.sendBeaconFinishedMsg(){
    bool rv = FALSE;
call Transport.sendBeaconFinishedMsg(&m_msg);
rv = SUCCESS;

return rv;
}

command result_t AppFunction.sendTransferMsg(
    uint16_t a_receiver
    , uint16_t a_receiver_counter
    , int8_t a_rssi
    , int8_t a_lqi
    , uint16_t a_sender
    , uint16_t a_sender_counter
    , uint8_t a_tx_power
){
    TransferMsg_t *msg = (TransferMsg_t*) m_msg.data;
    bool rv = FALSE;
msg->from = a_receiver;
msg->receiver_counter = a_receiver_counter;
msg->rssi = a_rssi;
msg->lqi = a_lqi;
msg->sender = a_sender;
msg->sender_counter = a_sender_counter;
msg->tx_power = a_tx_power;
call Transport.sendTransferMsg(&m_msg);
rv = SUCCESS;
return rv;
}

//command result_t AppFunction.sendTransferFinishedMsg(uint16_t a_beacons_number){
TransferFinishedMsg_t *msg = (TransferFinishedMsg_t*) m_msg.data;
bool rv = FALSE;
msg->beacons_number = a_beacons_number;
call Transport.sendTransferFinishedMsg(&m_msg);
rv = SUCCESS;
return rv;
}

//command void AppFunction.wait(uint16_t a_delay){
call Timer.start(TIMER_ONE_SHOT, a_delay);
}

//event result_t Timer.fired(){
signal AppEvent.onWaitDone();
return SUCCESS;
}

//command void AppFunction.blink(){
call BlinkTimer.start(TIMER_ONE_SHOT, 300);
call Leds.redOn();
}

//event result_t BlinkTimer.fired(){
call Leds.redOff();
return SUCCESS;
}

//command AppData* AppFunction.data(){
return &m_data;
}

//command void AppFunction.clearStorage(){
m_storage_length = 0;
}

//command uint16_t AppFunction.countSamples(){
return m_storage_length;
}

//command void AppFunction.storeSample(Sample *a_sample){
if (m_storage_length < sizeof(m_storage)/sizeof(m_storage[0])){
    Sample *s = &m_storage[m_storage_length];
s->sender = a_sample->sender;
}
s->counter = a_sample->counter;
s->tx_power = a_sample->tx_power;
s->rssi = a_sample->rssi;
s->lqi = a_sample->lqi;
m_storage_length += 1;
}

());//command void AppFunction.setTransmitPower(uint16_t a_tx_power){
if ((a_tx_power >= 1) && (a_tx_power <= 31)) {
call RadioControl.SetRFPower(RESOURCE_NONE, a_tx_power);
}

}//command bool AppFunction.loadSample(
  uint16_t a_index, Sample *rv_sample){
  bool rv = FALSE;
  if (a_index < m_storage_length ) {
    Sample *s = &(m_storage[a_index]);
    rv_sample->sender = s->sender;
    rv_sample->counter = s->counter;
    rv_sample->tx_power = s->tx_power;
    rv_sample->rssi = s->rssi;
    rv_sample->lqi = s->lqi;
    rv = TRUE;
  }
  return rv;
}

//======================================================================
// Interface Transport.
//======================================================================
//event void Transport.onSendDone(bool a_success){
  signal AppEvent.onSendDone(a_success);
}

//======================================================================
//event void Transport.onReceiveResetMsg(TOS_MsgPtr msg){
  signal AppEvent.onReset();
}

//======================================================================
//event void Transport.onReceiveBeaconStartMsg(TOS_MsgPtr msg){
  BeaconStartMsg_t *cmd = (BeaconStartMsg_t*)msg->data;
  signal AppEvent.onBeaconStart(cmd->sender,
      cmd->beacons_number, cmd->tx_power, cmd->delay);
}

//======================================================================
//event void Transport.onReceiveBeaconMsg(TOS_MsgPtr msg){
  BeaconMsg_t *cmd = (BeaconMsg_t*)msg->data;
  signal AppEvent.onBeacon(
    cmd->sender, cmd->counter, cmd->tx_power, msg->strength, msg->lqi);
}

//======================================================================
//event void Transport.onReceiveTransferStartMsg(TOS_MsgPtr msg){
  TransferStartMsg_t *cmd = (TransferStartMsg_t*)msg->data;
  signal AppEvent.onTransferStart(cmd->from);
}

//======================================================================
Listing A.3: AppEvent.nc - Application events declaration

```c
interface AppEvent {
  event void onReset();
  event void onBeaconStart(uint16_t a_mote, uint16_t a_beacons_number,
    uint8_t tx_power, uint16_t delay);
  event void onBeacon(uint16_t a_sender, uint16_t a_counter, uint8_t a_tx_power,
    uint8_t a_rssi, uint8_t lqi);
  event void onTransferStart(uint16_t a_mote);
  event void onSendDone(bool a_success);
  event void onWaitDone();
}
```

Listing A.4: AppFunction.nc - Application operations declaration

```c
typedef struct AppData {
  uint16_t counter;
  uint8_t tx_power;
  uint16_t delay;
} AppData;

typedef struct Sample {
  uint16_t sender;
  uint16_t counter;
  uint8_t tx_power;
  uint8_t rssi;
  uint8_t lqi;
} Sample;

interface AppFunction {
  command result_t sendBeaconMsg(
    uint16_t a_sender,
    uint16_t a_counter,
    uint8_t tx_power
  );

  command result_t sendBeaconFinishedMsg();

  command result_t sendTransferMsg(
    uint16_t a_receiver,
    uint16_t a_receiver_counter,
    uint8_t a_rssi,
    uint8_t a_lqi,
    uint16_t a_sender,
    uint16_t a_sender_counter,
    uint8_t a_tx_power
  );

  command result_t sendTransferFinishedMsg(uint16_t a_beacons_number);

  // Transmit power should be from 1 to 31 where
  // 1 correspond to minimal power (-25dBm)
  // 31 correspond to maximal power (0dBm)
  command void setTransmitPower(uint16_t a_tx_power);

  command void wait(uint16_t a_delay);
  command void blink();

  command AppData* data();

  command void clearStorage();
  command uint16_t countSamples();
  command void storeSample(Sample *a_sample);
  command bool loadSample(uint16_t a_index, Sample *a_sample);
}
```
Listing A.5: StateEvent.nc - State machine events declaration

```java
interface StateEvent {
  event void onStart();
  event void onIn();
  event void onOut();
}
```

Listing A.6: StateMachine.nc - State machine engine operations declaration

```java
typedef uint8_t IdState;

interface StateMachine {
  command void start(IdState a_state);
  command void changeState(IdState a_state);
  command IdState currentState();
}
```

Listing A.7: StateMachineM.nc - State machine engine implementation

```java
module StateMachineM{
  provides {
    interface StateMachine;
    interface StateEvent[IdState a_id];
    interface AppEvent[IdState a_id];
  }
  uses {
    interface AppEvent as App;
  }
  implementation {
    IdState m_state;
    IdState m_next_state;
    // Interface StateMachine:
    // ChangeStateTask:
    void task changeStateTask() {
      signal StateEvent.onOut[m_state]();
      m_state = m_next_state;
      signal StateEvent.onIn[m_state]();
    }
    // ChangeState:
    command void StateMachine.changeState(IdState a_state) {
      m_next_state = a_state;
      m_state = m_next_state;
      signal StateEvent.onOut[m_state]();
      m_state = m_next_state;
      signal StateEvent.onIn[m_state]();
    }
    // CurrentState:
    command IdState StateMachine.currentState() {
      return m_state;
    }
  }
}
```
// Interface StateEvent:
//----------------------------------------------------------------------
default event void StateEvent.onStart(IdState a_state) {}
//----------------------------------------------------------------------
default event void StateEvent.onIn(IdState a_state) {}
//----------------------------------------------------------------------
default event void StateEvent.onOut(IdState a_state) {}
//----------------------------------------------------------------------
// Interface AppEvent:
//----------------------------------------------------------------------
default event void AppEvent.onReset(IdState id) {};
//----------------------------------------------------------------------
default event void AppEvent.onBeaconStart(IdState id)(
    uint16_t a_mote, uint16_t a_beacons_number
    , uint8_t a_tx_power, uint16_t a_delay) {};
//----------------------------------------------------------------------
default event void AppEvent.onBeacon(IdState id)(
    uint16_t a_sender, uint16_t a_counter
    , uint8_t a_tx_power
    , uint8_t a_rssi, uint8_t a_lqi
 ) {};
//----------------------------------------------------------------------
default event void AppEvent.onTransferStart(IdState id)(
    uint16_t a_mote) {};
//----------------------------------------------------------------------
default event void AppEvent.onSendDone(IdState id)(
    bool a_success) {};
//----------------------------------------------------------------------
default event void AppEvent.onWaitDone(IdState id)();
//----------------------------------------------------------------------
// Interface App:
//----------------------------------------------------------------------
event void App.onReset() {
    signal AppEvent.onReset[m_state]();
}
//----------------------------------------------------------------------
event void App.onBeaconStart(
    uint16_t a_mote, uint16_t a_beacons_number
    , uint8_t a_tx_power, uint16_t a_delay) {
    signal AppEvent.onBeaconStart[m_state](
        a_mote, a_beacons_number, a_tx_power, a_delay);
}
//----------------------------------------------------------------------
event void App.onBeacon(
    uint16_t a_sender, uint16_t a_counter
    , uint8_t a_tx_power
    , uint8_t a_rssi, uint8_t a_lqi
 ) {
    signal AppEvent.onBeacon[m_state](
        a_sender, a_counter, a_tx_power
        , a_rssi, a_lqi
    );
}
//----------------------------------------------------------------------
event void App.onTransferStart(uint16_t a_mote) {
    signal AppEvent.onTransferStart[m_state](a_mote);
}
event void App.onSendDone(bool a_success) {
    signal AppEvent.onSendDone[m_state](a_success);
}

Listing A.8: Behavior.nc - Behavior interface declaration

interface Behavior {
    command void start();
}

Listing A.9: BehaviorC.nc - Behavior state machine wiring configuration

typedef enum {
    STATE_WAIT = unique("State")
    , STATE_BEACON = unique("State")
    , STATE_SAMPLE = unique("State")
    , STATE_TRANSFER = unique("State")
} StateEnum;

class BehaviorC {
    provides interface Behavior;
    uses interface AppEvent;
}

implementation {
    components BehaviorM
        , ExperimentM
        , StateMachineM
        , StateWaitP
        , StateBeaconP
        , StateSampleP
        , StateTransferP
        , LedsC
        , LoggerC
    
    StateMachineM.App = AppEvent;
    Behavior = BehaviorM.Behavior;
    BehaviorM.StateMachine -> StateMachineM.StateMachine;
    StateWaitP.StateMachine -> StateMachineM.StateMachine;
    StateWaitP.StateEvent -> StateMachineM.StateEvent[STATE_WAIT];
    StateWaitP.AppEvent -> StateMachineM.AppEvent[STATE_WAIT];
    StateWaitP.AppFunction -> ExperimentM.AppFunction;
    StateWaitP.Leds -> LedsC;
    StateWaitP.Logger -> LoggerC;

    StateBeaconP.StateMachine -> StateMachineM.StateMachine;
    StateBeaconP.StateEvent -> StateMachineM.StateEvent[STATE_BEACON];
    StateBeaconP.AppEvent -> StateMachineM.AppEvent[STATE_BEACON];
    StateBeaconP.AppFunction -> ExperimentM.AppFunction;
    StateBeaconP.Leds -> LedsC;
    StateBeaconP.Logger -> LoggerC;
StateSampleP.StateMachine -> StateMachineM.StateMachine;
StateSampleP.StateEvent -> StateMachineM.StateEvent[STATE_SAMPLE];
StateSampleP.AppFunction -> ExperimentM.AppFunction;
StateSampleP.Leds -> LedsC;
StateSampleP.Logger -> LoggerC;

StateTransferP.StateMachine -> StateMachineM.StateMachine;
StateTransferP.StateEvent -> StateMachineM.StateEvent[STATE_TRANSFER];
StateTransferP.AppFunction -> ExperimentM.AppFunction;
StateTransferP.Leds -> LedsC;
StateTransferP.Logger -> LoggerC;

Listing A.10: BehaviorM.nc - Behavior operations implementation

module BehaviorM {
  provides interface Behavior;
  uses {
    interface StateMachine;
  }
  implementation {
    command void Behavior.start() {
      call StateMachine.start(STATE_WAIT);
    }
  }
}

Listing A.11: StateBeaconP.nc - StateBeacon behavior implementation

module StateBeaconP {
  uses {
    interface StateMachine;
    interface StateEvent;
    interface AppEvent;
    interface AppFunction;
    interface Leds;
    interface Logger;
  }
  implementation {
    uint16_t m_cur;
    uint16_t m_total;
    uint16_t m_delay;
    uint8_t m_tx_power;
    // Interface StateEvent.
    event void StateEvent.onStart(){}
    // End of StateEvent.
    event void StateEvent.onIn() {
      AppData *data = call AppFunction.data();
      call Leds.redOff();
      call Leds.greenOff();
      call Leds.yellowOn();
      m_cur = 0;
    }
  }
}
m_total = data->counter;
m_delay = data->delay;
m_tx_power = data->tx_power;
call AppFunction.wait(m_delay);
}

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.

// Interface AppEvent.
Listing A.12: StateSampleP.nc - StateSample behavior implementation

```plaintext
define StateSampleP {
  uses {
    interface StateMachine;
    interface StateEvent;
    interface AppEvent;
    interface AppFunction;
    interface Leds;
    interface Logger;
  }
}

define StateEvent {
  event void onStart() {}
  event void onIn() {
    call Leds.redOff();
    call Leds.greenOn();
    call Leds.yellowOff();
    call AppFunction.clearStorage();
  }
  event void onOut() {
    call Leds.redOff();
    call Leds.greenOff();
    call Leds.yellowOff();
  }
}

define AppEvent {
  event void onReset() {
    call StateMachine.changeState(STATE_WAIT);
  }
  event void onBeaconStart(uint16 a_mote, uint16 a_beacons_number, uint8 a_tx_power, uint16 a_delay) {} }
  event void onBeacon(uint16 a_sender, uint16 a_counter, uint8 a_tx_power, uint8 a_rssi, uint8 a_lqi) {
    call AppFunction.blink();
    m_sample.sender = a_sender;
    m_sample.counter = a_counter;
    m_sample.tx_power = a_tx_power;
    m_sample.rssi = a_rssi;
    m_sample.lqi = a_lqi;
    call AppFunction.storeSample(&m_sample);
  }
  event void onTransferStart(uint16 a_mote) {
    call StateMachine.changeState(STATE_TRANSFER);
  }
  event void onSendDone(bool a_success) {} }
  event void onWaitDone() {}
```
Listing A.13: StateTransferP.nc - StateTransfer behavior implementation

module StateTransferP {
    uses {
        interface StateMachine;
        interface StateEvent;
        interface AppConfig;
        interface AppConfig;
        interface AppConfig;
        interface AppConfig;
        interface AppConfig;
        interface AppConfig;
    }
}

implementation {
    uint16_t m_cur;
    uint16_t m_total;
    uint16_t m_delay;
    Sample m_sample;

    // Interface StateEvent.
    event void StateEvent.onStart() {}

    event void StateEvent.onIn() {
        call Leds.redOff();
        call Leds.greenOn();
        call Leds.yellowOn();

        m_cur = 0;
        m_delay = 100;
        m_total = call AppConfig.countSamples();
        call Logger.log("total", m_total);
        call AppConfig.wait(m_delay);
    }

    event void StateEvent.onOut() {
        call Leds.redOff();
        call Leds.greenOff();
        call Leds.yellowOff();
    }

    // Interface AppConfig.
    event void AppConfig.onReset() {
        call AppConfig.changeState(STATE_WAIT);
    }

    event void AppConfig.onBeaconStart(uint16_t a_mote, uint16_t a_beacons_number
        , uint8_t a_tx_power, uint16_t a_delay
    ) {}
Listing A.14: StateWaitP.nc - StateWait behavior implementation

```plaintext
module StateWaitP {
  uses {
    interface StateMachine;
    interface StateEvent;
    interface AppEvent;
    interface AppFunction;
    interface Leds;
    interface Logger;
  }
}

implementation {
  // StateEvent
  event void StateEvent.onStart() {}

  event void StateEvent.onIn() {
    call Leds.redOff();
    call Leds.greenOff();
    call Leds.yellowOff();
  }

  event void StateEvent.onOut() {
    call Leds.redOff();
    call Leds.greenOff();
    call Leds.yellowOff();
  }

  // AppEvent
  event void AppEvent.onReset() {
    call StateMachine.changeState(STATE_WAIT);
  }
}
```
event void AppEvent.onBeaconStart(
    uint16_t a_mote, uint16_t a_beacons_number
    , uint8_t a_tx_power, uint16_t a_delay) {
    AppData *data = call AppFunction.data();
    data->counter = a_beacons_number;
    data->tx_power = a_tx_power;
    data->delay = a_delay;
    if (a_mote == TOS_LOCAL_ADDRESS) {
        call StateMachine.changeState(STATE_BEACON);
    } else {
        call StateMachine.changeState(STATE_SAMPLE);
    }
}

Listing A.15: Transport.nc - Communication protocol interface declaration

interface Transport {
    command void reset();
    command void sendBeaconMsg(TOS_MsgPtr a_msg);
    command void sendBeaconFinishedMsg(TOS_MsgPtr a_msg);
    command void sendTransferMsg(TOS_MsgPtr a_msg);
    command void sendTransferFinishedMsg(TOS_MsgPtr a_msg);
    event void onSendDone(bool a_success);
    event void onReceiveResetMsg(TOS_MsgPtr a_msg);
    event void onReceiveBeaconStartMsg(TOS_MsgPtr a_msg);
    event void onReceiveBeaconMsg(TOS_MsgPtr a_msg);
    event void onReceiveTransferStartMsg(TOS_MsgPtr a_msg);
    event void onReceiveTransferFinishedMsg(TOS_MsgPtr a_msg);
}

Listing A.16: TransportC.nc - Communication protocol wiring configuration

includes ResetMsg;
includes BeaconStartMsg;
includes BeaconMsg;
includes TransferStartMsg;
includes BeaconFinishedMsg;
includes TransferMsg;
includes TransferFinishedMsg;
includes ConfirmationMsg;

configuration TransportC {
  provides interface Transport;
}

implementation {
  components TransportM
    , GenericComm
    , Main
    , LedsC
    , TimerC
    , LoggerC
    ;

  Main.StdControl -> TimerC.StdControl;
  Main.StdControl -> TransportM.StdControl;

  Transport = TransportM.Transport;

  TransportM.Leds -> LedsC;
  TransportM.Logger -> LoggerC;

  TransportM.ReceiveResetMsg -> GenericComm.ReceiveMsg[AM_RESETMSG];
  TransportM.ReceiveBeaconStartMsg -> GenericComm.ReceiveMsg[AM_BEACONSTARTMSG];
  TransportM.ReceiveBeaconMsg -> GenericComm.ReceiveMsg[AM_BEACONMSG];
  TransportM.ReceiveTransferStartMsg -> GenericComm.ReceiveMsg[AM_TRANSFERSTARTMSG];
  TransportM.ReceiveConfirmationMsg -> GenericComm.ReceiveMsg[AM_CONFIRMATIONMSG];

  TransportM.SendBeaconMsg -> GenericComm.SendMsg[AM_BEACONMSG];
  TransportM.SendBeaconFinishedMsg ->
    GenericComm.SendMsg[AM_BEACONFINISHEDMSG];
  TransportM.SendTransferMsg -> GenericComm.SendMsg[AM_TRANSFERMSG];
  TransportM.SendTransferFinishedMsg ->
    GenericComm.SendMsg[AM_TRANSFERFINISHEDMSG];

  TransportM.SendConfirmationMsg ->
    GenericComm.SendMsg[AM_CONFIRMATIONMSG];

  TransportM.RetransmitTimer -> TimerC.Timer[unique("Timer")];
}

Listing A.17: TransportM.nc - Communication protocol implementation

includes ResetMsg;
includes BeaconStartMsg;
includes BeaconMsg;
includes TransferStartMsg;

includes BeaconFinishedMsg;
includes TransferMsg;
includes TransferFinishedMsg;
includes ConfirmationMsg;

module TransportM{
  provides {
    interface Transport;
    interface StdControl;
  }
  uses {
    interface Leds;
    interface Logger;

    interface SendMsg as SendBeaconMsg;
    interface SendMsg as SendBeaconFinishedMsg;
    interface SendMsg as SendTransferMsg;
    interface SendMsg as SendTransferFinishedMsg;
    interface SendMsg as SendConfirmationMsg;
    interface SendMsg as SendBeaconStartMsg;
    interface SendMsg as SendBeaconFinishedMsg;
    interface SendMsg as SendTransferStartMsg;
    interface SendMsg as SendTransferFinishedMsg;
    interface SendMsg as SendConfirmationMsg;
  }
}
interface SendMsg as SendTransferFinishedMsg;
interface SendMsg as SendConfirmationMsg;
interface ReceiveMsg as ReceiveResetMsg;
interface ReceiveMsg as ReceiveBeaconStartMsg;
interface ReceiveMsg as ReceiveBeaconMsg;
interface ReceiveMsg as ReceiveTransferStartMsg;
interface ReceiveMsg as ReceiveConfirmationMsg;
interface Timer as RetransmitTimer;

implementation {
enum {
    GATEWAY_ADDRESS = 0,
    MAX_RESEND_COUNT = 5,
    RESEND_DELAY = 800,
    uint8_t m_confirmation_expected=0;
    uint8_t m_confirmation_send=0;
    bool m_is_sending = FALSE;
    bool m_send_success = FALSE;
    unsigned char m_resend_count=0;
    TOS_Msg m_msg_confirmation;
    TOS_MsgPtr m_msg_send = 0;
};

command result_t StdControl.init() {
call Transport.reset();
return SUCCESS;
}

command result_t StdControl.start() {
call Transport.reset();
return SUCCESS;
}

command result_t StdControl.stop() {
return SUCCESS;
}

bool on_message_received(uint16_t a_from, uint8_t a_confirmation, TOS_MsgPtr a_msg) {
__attribute__((C, spontaneous)) {
bool rv = FALSE;
ConfirmationMsg_t *confirmation_data = (ConfirmationMsg_t*) m_msg_confirmation.data;
if (a_msg->addr != TOS_LOCAL_ADDRESS) {
    return FALSE;
} else if (a_from != GATEWAY_ADDRESS) {
    return FALSE;
} else if (a_confirmation == m_confirmation_expected) {
    rv = TRUE;
    m_confirmation_expected = m_confirmation_expected? 0:1;
} else {
    rv = FALSE;
}
}
// Send confirmation message.
// I do not check status of confirmation sending, because
// if confirmation is lost - other node will retransmit.
confirmation_data->confirmation = 1 - m_confirmation_expected;
call SendConfirmationMsg.send(
    GATEWAY_ADDRESS //a_from
    , sizeof(ConfirmationMsg_t), &m_msg_confirmation
);
return rv;
}

// Interface ReceiveResetMsg.

#endif
m_is_sending = FALSE;
return SUCCESS;
}

//======================================================================
// Interface SendTransferMsg.
//======================================================================

event result_t SendTransferMsg.sendDone(TOS_MsgPtr msg, result_t result) {
  m_is_sending = FALSE;
  return SUCCESS;
}

//======================================================================
// Interface SendTransferFinishedMsg.
//======================================================================

event result_t SendTransferFinishedMsg.sendDone(TOS_MsgPtr msg, result_t result) {
  m_is_sending = FALSE;
  return SUCCESS;
}

//======================================================================
// Interface SendConfirmationMsg.
//======================================================================

event result_t SendConfirmationMsg.sendDone(TOS_MsgPtr msg, result_t result) {
  m_is_sending = FALSE;
  return SUCCESS;
}

//======================================================================
// Interface Transport.
//======================================================================

command void Transport.sendBeaconMsg(TOS_MsgPtr a_msg) {
  call SendBeaconMsg.send(TOS_BCAST_ADDR, sizeof(BeaconMsg_t), a_msg);
  m_is_sending = FALSE;
}

command void Transport.reset() {
  call RetransmitTimer.stop();
  m_confirmation_expected = 0;
  m_confirmation_send = 0;
  m_is_sending = FALSE;
  m_send_success = FALSE;
  m_resend_count = 0;
  m_msg_send = 0;
}

void signalSendDone(bool a_success) __attribute__((C, spontaneous)){
  call RetransmitTimer.stop();
  m_resend_count = 0;
  if (a_success) {
    m_confirmation_send = m_confirmation_send? 0:1;
    call Logger.log("sendOk", 10);
  } else {
    call Logger.log("sendFalse", 20);
  }
  signal Transport.onSendDone(a_success);
}

void resend() __attribute__((C, spontaneous)) {
  if (m_msg_send == 0) {
    return;
  }
  switch (m_msg_send->type) {
case AM_BEACONFINISHEDMSG:
    call SendBeaconFinishedMsg.send(GATEWAY_ADDRESS, sizeof(BeaconFinishedMsg_t), m_msg_send);
    break;

case AM_TRANSFERMSG:
    call SendTransferMsg.send(GATEWAY_ADDRESS, sizeof(TransferMsg_t), m_msg_send);
    break;

case AM_TRANSFERFINISHEDMSG:
    call SendTransferFinishedMsg.send(GATEWAY_ADDRESS, sizeof(TransferFinishedMsg_t), m_msg_send);
    break;

}  
m_resend_count++;  
call RetransmitTimer.start(TIMER_ONE_SHOT, RESEND_DELAY);  
}

command void Transport.sendBeaconFinishedMsg(TOS_MsgPtr a_msg) {
    if (m_msg_send != 0) {
        call Logger.log("busy", '1');
        signalSendDone(FALSE);
    } else {
        BeaconFinishedMsg_t *msg_data = (BeaconFinishedMsg_t*) a_msg->data;
        // Fill message parameters.
        m_resend_count = 0;
        m_msg_send = a_msg;
        m_msg_send->type = AM_BEACONFINISHEDMSG;
        msg_data->confirmation = m_confirmation_send;
        msg_data->from = TOS_LOCAL_ADDRESS;
        // Start retransmit task.
        resend();
    }
}

command void Transport.sendTransferMsg(TOS_MsgPtr a_msg) {
    if (m_msg_send != 0) {
        call Logger.log("busy", '2');
        signalSendDone(FALSE);
    } else {
        TransferMsg_t *msg_data = (TransferMsg_t*) a_msg->data;
        // Fill message parameters.
        m_resend_count = 0;
        m_msg_send = a_msg;
        m_msg_send->type = AM_TRANSFERMSG;
        msg_data->confirmation = m_confirmation_send;
        msg_data->from = TOS_LOCAL_ADDRESS;
        // Start retransmit task.
        resend();
    }
}

command void Transport.sendTransferFinishedMsg(TOS_MsgPtr a_msg) {
    if (m_msg_send != 0) {
        call Logger.log("busy", 3);
        signalSendDone(FALSE);
    } else {
        TransferFinishedMsg_t *msg_data = (TransferFinishedMsg_t*) a_msg->data;
// Fill message parameters.
    m_resend_count = 0;
    m_msg_send = a_msg;
    m_msg_send->type = AM_TRANSFERFINISHEDMSG;
    msg_data->confirmation = m_confirmation_send;
    msg_data->from = TOS_LOCAL_ADDRESS;

    // Start retransmit task.
    resend();
}

//======================================================================
// Interface RetransmitTimer.
//----------------------------------------------------------------------
event result_t RetransmitTimer.fired() {
    if (m_resend_count >= MAX_RESEND_COUNT) {
        call Logger.log("sendFail", m_resend_count);
        signalSendDone(FALSE);
    }
    else {
        resend();
    }
    return SUCCESS;
}

//======================================================================
// Interface ReceiveConfirmationMsg.
//----------------------------------------------------------------------
event TOS_MsgPtr ReceiveConfirmationMsg.receive(TOS_MsgPtr a_msg) {
    ConfirmationMsg_t *data = (ConfirmationMsg_t *) a_msg->data;
    if (a_msg->addr != TOS_LOCAL_ADDRESS) {
        call Logger.log("notMe", a_msg->addr);
        call Logger.log("nmFrom", data->from);
    }
    else if (m_msg_send == 0) {
        call Logger.log("nosend", 1);
    }
    else if (data->from != GATEWAY_ADDRESS) {
        call Logger.log("nogtw", data->from);
    }
    else if (data->confirmation != m_confirmation_send) {
        call Logger.log("wrongAck", data->confirmation);
    }
    else {
        call Logger.log("goodAck", data->confirmation);
        call Logger.log("sd_ack", m_msg_send->type);
        signalSendDone(TRUE);
    }
    return a_msg;
}

Listing A.18: ResetMsg.h - ResetMsg declaration

enum{
    AM_RESETMSG = 10,
};
typed struct ResetMsg
{
    uint8_t confirmation;
    uint16_t from;
} ResetMsg_t;
Listing A.19: BeaconStartMsg.h - BeaconStartMsg declaration

```c
enum {
    AM_BEACONSTARTMSG = 21,
};
typedef struct BeaconStartMsg{
    // Acknowledgement fields.
    uint8_t confirmation;
    uint16_t from;
    // Sender mote's id to start sending beacons.
    uint16_t sender;
    // Number of beacons to send.
    uint16_t beacons_number;
    // Transmit power.
    uint8_t tx_power;
    // Delay between beacons.
    uint16_t delay;
} BeaconStartMsg_t;
```

Listing A.20: BeaconMsg.h - BeaconMsg declaration

```c
enum {
    AM_BEACONMSG = 22,
};
typedef struct BeaconMsg{
    // Mote's id to start sending beacons.
    uint16_t sender;
    // Number of current beacon in a burst.
    uint16_t counter;
    // Transmission power.
    uint8_t tx_power;
} BeaconMsg_t;
```

Listing A.21: BeaconFinishedMsg.h - BeaconFinishedMsg declaration

```c
enum {
    AM_BEACONFINISHEDMSG = 23,
};
typedef struct BeaconFinishedMsg{
    // Confirmation
    uint8_t confirmation;
    // Id of sender mote's which finished sending beacons.
    uint16_t from;
} BeaconFinishedMsg_t;
```

Listing A.22: TransferStartMsg.h - TransferStartMsg declaration
```c
enum {
    AM_TRANSFERSTARTMSG = 31,
};

typedef struct TransferStartMsg{
    uint8_t confirmation;
    // Mote's id to start transferring sample data.
    uint16_t from;
} TransferStartMsg_t;

Listing A.23: TransferMsg.h - TransferMsg declaration
```

```c
enum {
    AM_TRANSFERMSG = 32,
};

typedef struct TransferMsg{
    uint8_t confirmation;
    // Id of the node who receive message.
    uint16_t from;
    // Receive order message countr.
    uint16_t receiver_counter;
    // RSSI value
    uint8_t rssi;
    // LQI value
    uint8_t lqi;
    // Id of the node who sent message.
    uint16_t sender;
    // Send order message counter.
    uint16_t sender_counter;
    // Transmission power.
    uint8_t tx_power;
} TransferMsg_t;

Listing A.24: TransferFinishedMsg.h - TransferFinishedMsg declaration
```

```c
enum {
    AM_TRANSFERFINISHEDMSG = 33,
};

typedef struct TransferFinishedMsg{
    uint8_t confirmation;
    // Id of the receiver mote which finished transfer.
    uint16_t from;
    // Number of beacons to send.
    uint16_t beacons_number;
} TransferFinishedMsg_t;
```
Listing A.25: ConfirmationMsg.h - ConfirmationMsg declaration

```c
enum {
  AM_CONFIRMATIONMSG = 77,
};

typedef struct ConfirmationMsg{
  // Delay between beacons.
  uint8_t confirmation;
  // Id of mote who sent confirmation
  uint16_t from;
} ConfirmationMsg_t;
```

Listing A.26: LogMsg.h - LogMsg declaration

```c
enum {
  AM_LOGMSG = 7,
};

typedef struct LogMsg {
  int8_t id; // Message type.
  char text[12]; // Message text.
  int16_t value; // Message value.
} LogMsg_t;
```

Listing A.27: CircularQueue.h - Circular queue implementation

```c
#ifndef __CIRCULAR_QUEUE_H__
#define __CIRCULAR_QUEUE_H__

typedef uint8_t QueueSize;

// Invariants:
// 1. 0 <= front <= size
// 2. 0 <= size <= size
// 3. If queue is empty, front == back == size.
typedef struct {
  // Index of the item which was put in the queue first.
  QueueSize front;
  // Index of the item which was put in the queue last
  QueueSize back;
  // Size of the buffer managed by queue.
  QueueSize size;
} Queue;

// Initialises internal state.
void queueInit(Queue* const me, QueueSize a_size);

// Check if there are any items in a queue.
bool queueIsEmpty(Queue* const me);

// Check if there is a place for putting new items into the queue.
bool queueIsFull(Queue* const me);

// Put item into the queue.
bool queuePush(Queue* const me);

// Remove item from the queue.
void queuePop(Queue* const me);
```
// Tell how many items in a queue.
QueueSize queueLength(Queue* const me);

// Setup internal state of the queue.
// me: Pointer to the queue internal data structure (Not NULL).
// a_size: Size of the external buffer managed by the queue.
void queueInit(Queue* const me, QueueSize a_size) {
    me->front = a_size;
    me->back = a_size;
    me->size = a_size;
}

// Return TRUE if queue does not store any items and false otherwise.
bool queueIsEmpty(Queue* const me) {
    return ( (me->front == me->size) && (me->back == me->size) ) ? TRUE : FALSE;
}

// Tell how many items in a queue.
QueueSize queueLength(Queue* const me) {
    QueueSize rv = 0;
    if (!queueIsEmpty(me)) {
        rv = (me->back >= me->front) ? me->back - me->front + 1 : me->size - (me->front - me->back - 1);
    }
    return rv;
}

// Return TRUE if queue does not store any items and false otherwise.
bool queueIsFull(Queue* const me) {
    return ( queueLength(me) == me->size ) ? TRUE : FALSE;
}

// Put item into the queue.
// If operation was successful, me->back now points to the next item.
// Otherwise it stays at the same place.
// Return True if new item was put in a queue. False otherwise.
bool queuePush(Queue* const me) {
    bool rv = FALSE;
    if (queueIsEmpty(me)) {
        me->front = 0;
        me->back = 0;
        rv = TRUE;
    } else if (!queueIsFull(me)) {
        me->back++;
        me->back = (me->back == me->size)? 0: me->back;
        rv = TRUE;
    }
    return rv;
}

// Remove item from the queue.
void queuePop(Queue* const me) {
    if ( me->front == me->back ) { // if last item...
        // make queue empty...
        me->front = me->size;
        me->back = me->size;
    } else {
        me->front++;
        me->front = (me->front == me->size)? 0: me->front;
    }
Listing A.28: Logger.nc - Debug logger interface declaration

```c
interface Logger {
    command void log_value(int16_t a_value);
    command void log(const char* a_str, int16_t a_value);
}
```

Listing A.29: LoggerC.nc - Debug logger wiring configuration

```c
configuration LoggerC {
    provides {
        interface Logger;
        interface StdControl;
    }
}
```

Listing A.30: LoggerM.nc - Debug logger implementation

```c
includes LogMsg;
includes CircularQueue;

module LoggerM {
    provides interface StdControl;
    provides interface Logger;
    uses interface SendMsg;
}
```

```c
// Shared variables:
bool m_sending;
TOS_Msg m_msg;

enum {BUFFER_SIZE = 16};
LogMsg_t m_buf[BUFFER_SIZE];
Queue m_queue;

//======================================================================
// Interface StdControl.
//----------------------------------------------------------------------
command result_t StdControl.init() {
    m_sending = FALSE;
    queueInit(&m_queue, BUFFER_SIZE);
    return SUCCESS;
}
```
command result_t StdControl.start() {
    m_sending = FALSE;
    return SUCCESS;
}

command result_t StdControl.stop() {
    return SUCCESS;
}

void task taskSend() {
    LogMsg_t *data = (LogMsg_t*) m_msg.data;
    if (!queueIsEmpty(&m_queue)) {
        unsigned short i;
        LogMsg_t *msg = &(m_buf[m_queue.front]);
        data->id = 1;
        data->text[0] = 0;
        for (i = 0; i < sizeof(data->text)/sizeof(data->text[0]); i++) {
            data->text[i] = msg->text[i];
            if (data->text[i] == 0)
                break;
        }
        data->value = msg->value;
        if ( !m_sending && \call SendMsg.send(TOS_UART_ADDR, sizeof(LogMsg_t), \&m_msg) ) {
            m_sending = TRUE;
        }
        else {
            post taskSend();
        }
    }
}

//======================================================================
// Interface SendMsg.
//----------------------------------------------------------------------
event result_t SendMsg.sendDone(TOS_MsgPtr a_msg, result_t a_success) {
    m_sending = FALSE;
    if (a_success) {
        queuePop(&m_queue);
    }
    if (!queueIsEmpty(&m_queue)) {
        post taskSend();
    }
    return SUCCESS;
}

//======================================================================
// Interface Logger.
//----------------------------------------------------------------------
command void Logger.log_value(int16_t a_value) {
    if ( queuePush(&m_queue) ) {
        LogMsg_t *msg = &(m_buf[m_queue.back]);
        msg->value = a_value;
        post taskSend();
    }
}

command void Logger.log(const char *a_str, int16_t a_value) {
    if ( queuePush(&m_queue) ) {
        unsigned short i = 0;
        LogMsg_t *msg = &(m_buf[m_queue.back]);
        msg->value = a_value;
for (i = 0; i < sizeof(msg->text)/sizeof(msg->text[0]);
    i++) {
    msg->text[i] = a_str[i];
    if (msg->text[i] == 0) {
        break;
    }
}
post taskSend();
Appendix B

Collected experiment measurements

This annex contains means and deviations of all RSSI measurements collected during all experiments.
### Table B.1: Experiment 1. Means and standard deviations of RSSI measurements.

<table>
<thead>
<tr>
<th>D (m)</th>
<th>TX=2 mean</th>
<th>TX=3 mean</th>
<th>TX=4 mean</th>
<th>TX=5 mean</th>
<th>TX=9 mean</th>
<th>TX=10 mean</th>
<th>TX=11 mean</th>
<th>TX=19 mean</th>
<th>TX=20 mean</th>
<th>TX=21 mean</th>
<th>TX=29 mean</th>
<th>TX=30 mean</th>
<th>TX=31 mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.5</td>
<td>210.6</td>
<td>227.8</td>
<td>235.1</td>
<td>234.8</td>
<td>240</td>
<td>240</td>
<td>242.5</td>
<td>247.9</td>
<td>219.4</td>
<td>249.4</td>
<td>248.8</td>
<td>162.1</td>
<td>120.3</td>
</tr>
<tr>
<td>1</td>
<td>222.5</td>
<td>225.8</td>
<td>228.7</td>
<td>229.1</td>
<td>236.2</td>
<td>236.1</td>
<td>238.2</td>
<td>243.4</td>
<td>241.5</td>
<td>249.2</td>
<td>247.5</td>
<td>246.3</td>
<td>247.2</td>
</tr>
<tr>
<td>1.5</td>
<td>227.5</td>
<td>235</td>
<td>235</td>
<td>235</td>
<td>241</td>
<td>241</td>
<td>244.2</td>
<td>241.7</td>
<td>248.9</td>
<td>248.9</td>
<td>253.9</td>
<td>254</td>
<td>254.7</td>
</tr>
<tr>
<td>2</td>
<td>230.9</td>
<td>235</td>
<td>237</td>
<td>234.5</td>
<td>234.6</td>
<td>234.6</td>
<td>236.4</td>
<td>242.7</td>
<td>243.1</td>
<td>241.2</td>
<td>241.2</td>
<td>241.2</td>
<td>245.5</td>
</tr>
<tr>
<td>2.5</td>
<td>215.9</td>
<td>222</td>
<td>222.2</td>
<td>229.4</td>
<td>229.9</td>
<td>232</td>
<td>237</td>
<td>243.2</td>
<td>241.4</td>
<td>241.4</td>
<td>243.2</td>
<td>243.2</td>
<td>243.4</td>
</tr>
<tr>
<td>3</td>
<td>224.4</td>
<td>232.3</td>
<td>236.8</td>
<td>230.2</td>
<td>230.3</td>
<td>232.8</td>
<td>237</td>
<td>243.1</td>
<td>242.2</td>
<td>242.2</td>
<td>245.9</td>
<td>245.5</td>
<td>246.1</td>
</tr>
<tr>
<td>3.5</td>
<td>221</td>
<td>227.9</td>
<td>228.2</td>
<td>235.1</td>
<td>235</td>
<td>237</td>
<td>243.2</td>
<td>242.2</td>
<td>242.2</td>
<td>245.9</td>
<td>245.5</td>
<td>246.1</td>
<td>246.4</td>
</tr>
<tr>
<td>4</td>
<td>217.9</td>
<td>220</td>
<td>230.2</td>
<td>227</td>
<td>225.8</td>
<td>228.3</td>
<td>234.4</td>
<td>236.8</td>
<td>238.3</td>
<td>241.9</td>
<td>241.6</td>
<td>241.6</td>
<td>244.3</td>
</tr>
<tr>
<td>5</td>
<td>217.7</td>
<td>224</td>
<td>221.1</td>
<td>230.6</td>
<td>231.2</td>
<td>232.6</td>
<td>238.2</td>
<td>239.4</td>
<td>239.4</td>
<td>245.1</td>
<td>244.7</td>
<td>244.7</td>
<td>244.7</td>
</tr>
<tr>
<td>6</td>
<td>200.6</td>
<td>215.4</td>
<td>215.9</td>
<td>222.4</td>
<td>223</td>
<td>224.5</td>
<td>229.7</td>
<td>231.9</td>
<td>228.6</td>
<td>235.1</td>
<td>236.5</td>
<td>236.5</td>
<td>236.9</td>
</tr>
<tr>
<td>7</td>
<td>213.5</td>
<td>203.6</td>
<td>221</td>
<td>227</td>
<td>227.4</td>
<td>229.1</td>
<td>235</td>
<td>236.4</td>
<td>236.4</td>
<td>236.4</td>
<td>236.4</td>
<td>241</td>
<td>243.2</td>
</tr>
<tr>
<td>7.5</td>
<td>212.7</td>
<td>219.6</td>
<td>220</td>
<td>226.9</td>
<td>226.9</td>
<td>229</td>
<td>234.4</td>
<td>235.1</td>
<td>238.8</td>
<td>238.9</td>
<td>238.9</td>
<td>239.5</td>
<td>239.5</td>
</tr>
<tr>
<td>8</td>
<td>211.6</td>
<td>218</td>
<td>218.2</td>
<td>224.8</td>
<td>224.5</td>
<td>225</td>
<td>235.3</td>
<td>226.3</td>
<td>226.3</td>
<td>224.9</td>
<td>229.7</td>
<td>231.7</td>
<td>231.7</td>
</tr>
<tr>
<td>12</td>
<td>212</td>
<td>212.2</td>
<td>217.5</td>
<td>223.1</td>
<td>223.5</td>
<td>224.8</td>
<td>225.4</td>
<td>226.2</td>
<td>226.2</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
</tr>
<tr>
<td>13</td>
<td>211</td>
<td>217.2</td>
<td>217.2</td>
<td>223.1</td>
<td>223.5</td>
<td>224.8</td>
<td>225.4</td>
<td>226.2</td>
<td>226.2</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
</tr>
<tr>
<td>14</td>
<td>206.6</td>
<td>211.1</td>
<td>217.1</td>
<td>217.4</td>
<td>217.4</td>
<td>217.9</td>
<td>225.1</td>
<td>226.6</td>
<td>226.6</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
</tr>
<tr>
<td>15</td>
<td>210.1</td>
<td>216.1</td>
<td>216</td>
<td>222.9</td>
<td>222.9</td>
<td>224.9</td>
<td>230.9</td>
<td>232.1</td>
<td>232.1</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
<td>231.7</td>
</tr>
<tr>
<td>16</td>
<td>203.1</td>
<td>213</td>
<td>213.2</td>
<td>220.2</td>
<td>220.5</td>
<td>222.4</td>
<td>228</td>
<td>229.6</td>
<td>231.1</td>
<td>234.9</td>
<td>233.2</td>
<td>233.2</td>
<td>234.5</td>
</tr>
</tbody>
</table>

Table B.1: Experiment 1. Means and standard deviations of RSSI measurements.
### Appendix B. Collected experiment measurements

Table B.2: Experiment 2. Means and standard deviations of RSSI measurements.

<table>
<thead>
<tr>
<th>D(m)</th>
<th>TX=3 mean</th>
<th>TX=3 SD</th>
<th>TX=5 mean</th>
<th>TX=5 SD</th>
<th>TX=10 mean</th>
<th>TX=10 SD</th>
<th>TX=20 mean</th>
<th>TX=20 SD</th>
<th>TX=31 mean</th>
<th>TX=31 SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>226.4</td>
<td>4.9</td>
<td>233.5</td>
<td>5.3</td>
<td>239.1</td>
<td>4.5</td>
<td>220.4</td>
<td>76.4</td>
<td>197.3</td>
<td>100.3</td>
</tr>
<tr>
<td>3</td>
<td>221.5</td>
<td>3.3</td>
<td>229.1</td>
<td>3.6</td>
<td>235.3</td>
<td>3.6</td>
<td>243.4</td>
<td>3.1</td>
<td>246.5</td>
<td>2.9</td>
</tr>
<tr>
<td>7</td>
<td>213.2</td>
<td>2.8</td>
<td>217.3</td>
<td>3.5</td>
<td>222.6</td>
<td>4.7</td>
<td>231.7</td>
<td>4.7</td>
<td>235.6</td>
<td>4.5</td>
</tr>
<tr>
<td>12</td>
<td>211.9</td>
<td>0.4</td>
<td>215.7</td>
<td>1.7</td>
<td>217.8</td>
<td>4.2</td>
<td>224.6</td>
<td>5.9</td>
<td>228.0</td>
<td>6.6</td>
</tr>
</tbody>
</table>

Table B.3: Experiment 3. Means and standard deviations of RSSI measurements (classroom).

<table>
<thead>
<tr>
<th>D(m)</th>
<th>TX=5 mean</th>
<th>TX=5 SD</th>
<th>TX=10 mean</th>
<th>TX=10 SD</th>
<th>TX=20 mean</th>
<th>TX=20 SD</th>
<th>TX=31 mean</th>
<th>TX=31 SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>231.3</td>
<td>6.2</td>
<td>237.7</td>
<td>5.8</td>
<td>233.5</td>
<td>54.3</td>
<td>193.1</td>
<td>102.5</td>
</tr>
<tr>
<td>2</td>
<td>228.1</td>
<td>5.5</td>
<td>234.4</td>
<td>6.7</td>
<td>242.4</td>
<td>4.4</td>
<td>233.8</td>
<td>49.2</td>
</tr>
<tr>
<td>3</td>
<td>227.9</td>
<td>4.5</td>
<td>234.4</td>
<td>4.6</td>
<td>242.4</td>
<td>3.1</td>
<td>245.9</td>
<td>3.6</td>
</tr>
<tr>
<td>5</td>
<td>228.0</td>
<td>3.7</td>
<td>234.8</td>
<td>3.4</td>
<td>243.2</td>
<td>2.6</td>
<td>246.1</td>
<td>2.9</td>
</tr>
<tr>
<td>6</td>
<td>221.3</td>
<td>5.3</td>
<td>228.3</td>
<td>5.7</td>
<td>237.5</td>
<td>5.3</td>
<td>239.8</td>
<td>4.6</td>
</tr>
<tr>
<td>8</td>
<td>218.2</td>
<td>4.3</td>
<td>220.4</td>
<td>6.2</td>
<td>228.7</td>
<td>6.8</td>
<td>232.5</td>
<td>7.4</td>
</tr>
<tr>
<td>9</td>
<td>221.1</td>
<td>2.8</td>
<td>224.9</td>
<td>7.0</td>
<td>233.9</td>
<td>6.9</td>
<td>240.1</td>
<td>4.3</td>
</tr>
<tr>
<td>10</td>
<td>220.8</td>
<td>2.8</td>
<td>226.0</td>
<td>3.7</td>
<td>235.1</td>
<td>3.6</td>
<td>238.4</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Table B.4: Experiment 4. Means and standard deviations of RSSI measurements (tunnel).

<table>
<thead>
<tr>
<th>D(m)</th>
<th>TX=5 mean</th>
<th>TX=5 SD</th>
<th>TX=10 mean</th>
<th>TX=10 SD</th>
<th>TX=20 mean</th>
<th>TX=20 SD</th>
<th>TX=31 mean</th>
<th>TX=31 SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>227.4</td>
<td>4.1</td>
<td>234.5</td>
<td>3.1</td>
<td>242.8</td>
<td>2.5</td>
<td>245.6</td>
<td>2.6</td>
</tr>
<tr>
<td>2</td>
<td>228.8</td>
<td>4.4</td>
<td>235.0</td>
<td>4.3</td>
<td>242.8</td>
<td>3.5</td>
<td>246.0</td>
<td>16.3</td>
</tr>
<tr>
<td>3</td>
<td>228.1</td>
<td>6.1</td>
<td>234.0</td>
<td>6.8</td>
<td>242.1</td>
<td>5.7</td>
<td>244.8</td>
<td>6.9</td>
</tr>
<tr>
<td>5</td>
<td>215.8</td>
<td>3.3</td>
<td>221.0</td>
<td>4.0</td>
<td>229.6</td>
<td>3.9</td>
<td>233.3</td>
<td>5.2</td>
</tr>
<tr>
<td>6</td>
<td>215.8</td>
<td>3.6</td>
<td>220.9</td>
<td>4.8</td>
<td>229.7</td>
<td>4.9</td>
<td>233.0</td>
<td>5.6</td>
</tr>
<tr>
<td>8</td>
<td>216.1</td>
<td>2.3</td>
<td>220.7</td>
<td>3.6</td>
<td>228.2</td>
<td>5.2</td>
<td>232.2</td>
<td>6.4</td>
</tr>
<tr>
<td>9</td>
<td>214.1</td>
<td>3.1</td>
<td>218.9</td>
<td>3.9</td>
<td>225.7</td>
<td>5.4</td>
<td>229.7</td>
<td>5.8</td>
</tr>
<tr>
<td>10</td>
<td>213.6</td>
<td>2.6</td>
<td>217.8</td>
<td>3.6</td>
<td>225.3</td>
<td>4.4</td>
<td>230.0</td>
<td>4.4</td>
</tr>
</tbody>
</table>
Index

anchor-based algorithms, 33
anchor-free algorithms, 33
anchors, 25, 33
angulation, 28
Automatic Repeat Request, 51
beacons, 40
boxplot, 52
Centralized algorithms, 34
Cramér-Rao bound, 34
deployment, 40
deployment stage, 43
distributed algorithms, 34
edge detection, 15
event, 8
event classification, 9
event detection, 9
extreme outlier, 54
F random variable, 67
fences, 52
function approximation, 15
gateway, 8
implementation, 39
initialization stage, 43
interaction patterns, 8
introduction, 1
Lateration, 25
Location fingerprinting, 31
measuring stage, 44
mild outlier, 52
Multi-Dimensional Scaling, 29
multi-hop algorithms, 34
multi-hop communication, 8
nearest neighbor in signal space, 31
nuisance variation, 65
outliers, 52
path loss, 24
PdM, 11
periodic measurements, 11
Positioning, 25
Predictive maintenance, 11
randomized block design, 59
range-based algorithms, 34
Range-free algorithms, 34
Ranging, 22
RBF-$mq$, 69
Received Signal Strength Indicator, 24
receivers, 40
relay, 8
residual effects, 65
score, 59
sender, 40
SHM, 13
Single-hop algorithms, 33
sink, 8
sleep time, 12
source, 8
structural health monitoring, 13
Time Difference of Arrival, 23
Index

Time of Arrival, 23
tracking, 18
transfer stage, 44
treatment, 59
trial, 41

wireless sensor network, 1
WSN, 1