EXPERIMENTS FOR EVALUATING SENSORS' PRECISION IN WIRELESS SENSOR NETWORKS

Ronald Beaubrun and Siarhei Smolau

Department of Computer Science and Software Engineering,
Laval University, Quebec, Canada
Email: Ronald.Beaubrun@ift.ulaval.ca; Siarhei.Smolau.1@ulaval.ca

ABSTRACT

In this paper, we propose a set of experiments for evaluating sensors' precision in measuring received signal strength indicator (RSSI) in wireless sensor networks (WSN). More specifically, a WSN application is implemented for collecting RSSI measurements in different conditions. The application consists of two parts: an experiment control script which runs on a computer, and an experiment sensor firmware which runs on each WSN node. Statistical analysis of variance (ANOVA) is performed to determine if the WSN nodes used are manufactured with enough precision. Obtained results demonstrate that the differences between the sensors are insignificant, even if the RSSI measurements have significant variation.

KEYWORDS

Analysis of variance (ANOVA), received signal strength indicator (RSSI), wireless sensor networks (WSN).

1. INTRODUCTION

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 [1], [2], [3]. Such applications require that each sensor knows its exact location. In this context, a localization system is required to provide position information to the sensors. Such a system uses a method for distance evaluation [4], [5].

The received signal strength indicator (RSSI) is one of the simplest methods that has been used for estimation of distances between sensors [6], [7], [8]. The distance estimates are based on the strength of the signal received by another sensor. The main advantage of this method is its low cost, since most receivers are capable of estimating the received signal strength. However, several questions arise. What would be the transmission range for real devices running in real environments? Do RSSI measurements vary for different devices in the same conditions? Should each WSN node be calibrated separately in order to take into account the differences between individual sensors? Answers to all those questions affect the choice of the algorithm, requirements and performance of localization application.

This paper proposes a set of experiments that have been made for sensors' precision evaluation in terms of RSSI measurements, and presents results obtained from such experiments. It is organized as follows. Section 2 describes the experiment setup. Section 3 describes the implementation of the WSN application used to collect RSSI data. Section 4 presents and analyses the set of collected data. Section 5 gives some concluding remarks.

2. EXPERIMENT SETUP

2.1. Hardware and software platform

For the experiments, we are using *Tmote Sky* wireless sensor modules from Moteiv [9]. Each module is controlled by Texas Instruments MSP430 microcontroller which has 10 kB RAM, 48 kB Flash memory, and runs at 8 MHz. Also, each module provides integrated light, temperature and humidity sensors. For radio communication, each module uses a Chipcon CC2400 RF transceiver chip. This chip is compliant with IEEE 802.15.4 standard and allows to measure received signal strength. According to [10], the RSSI reading provides a measurement of the signal power entering the RF input. The scale is logarithmic, so that *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:

$$P = RSSI\_VAL + RSSI\_OFFSET\ [dBm] \tag{1}$$

The nominal value of *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 sensor, controlled by *TinyOS* [11]. 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* [12], 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.

2.2. Requirements and deployment

The requirements for the WSN application are specified in accordance with the planned experiment design. First, it should be possible to change the experiment parameters, *i.e.*, the transmission power, the number and coordinates of sensors, the number of beacons sent by each sender. Second, deployments are expensive. In this context, it is needed to place sensors 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 sensors may become inaccessible during an experiment (in case of battery failure). Thus, the application should be robust against missing messages and single sensor failure. In other words, data from the working sensors should be downloaded, and each experiment should be able to continue with the working sensors.

A WSN application is a distributed program which consists of several modules that are executed on different computers. Such modules are illustrated in the deployment diagram presented in Figure 1. In such a diagram, the 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 sensor 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 sensors distribution. It reads data from the gateway sensor via USB, detects when a complete message from a node is available and forwards it to any other application using TCP socket. The experiment control script is written in python in order to control the experiments.

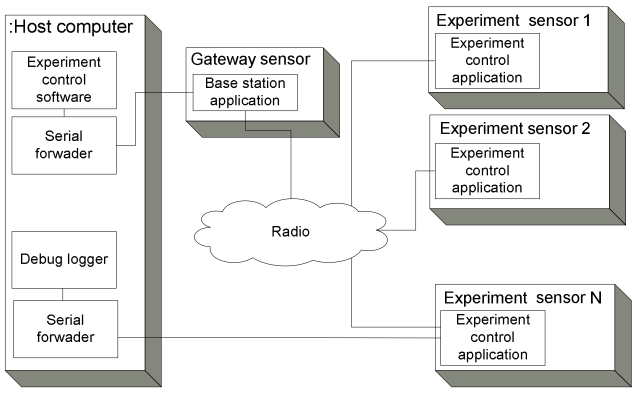

Figure 1. Deployment diagram

The gateway sensor 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 node executes an experiment application. There is only one kind of experiment application which allows a sensor to play the role of sender or receiver, depending on the commands received from the application control script. Each sensor has a unique address. The experiment control script knows the addresses of all sensors in the network. There is no multi-hop communication, *i.e.*, every sensor sends messages to the gateway sensor directly.

2.3. Application behavior

The application behavior is represented by the activity diagram illustrated in Figure 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 sensor.

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

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

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

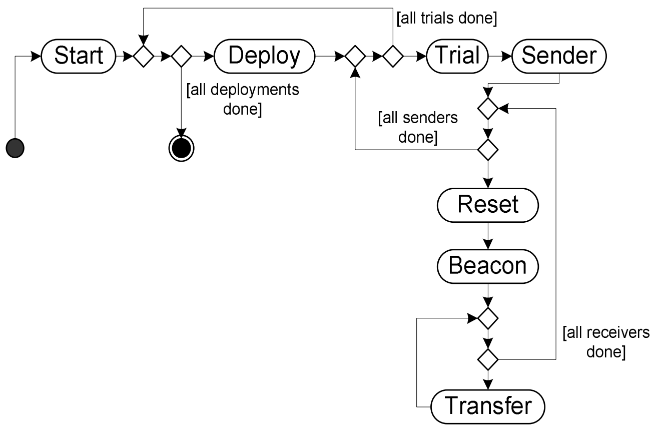

Figure 2. Activity diagram

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

```
<scenario>
        <deployment>
                <trialbeacons_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"/>
                <mote id="1"  x="0"  y="0"  z="0" />
                <mote id="2"  x="2"  y="0"  z="0" />
        </deployment>
</scenario>
```

Each deployment is specified by the number of sensors and their coordinates, as well as the number and parameters of the trials. Each sensor has a network address and its coordinates. Each trial specifies the number of beacons to be sent by the sender, as well as the number, delay between beacons and transmitted power.

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

```
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 sensors 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 computation performed by each sensor is specified as a response to external and internal events.

3. Implementation Parameters

3.1. Implementation of the communication protocol

There are 9 types of messages exchanged between the sensors and the host computer. Among them, 3 types are used to allow the application to measure RSSI values. They are denoted: *BeaconStartMsg*, *BeaconMsg* and *BeaconFinishedMsg*. Also, 3 types of messages are needed to allow the application to collect measured RSSI data. They are denoted: *TransferStartMsg*, *TransferMsg* and *TransferFinishedMsg*. Moreover, 3 types of messages are used to allow debugging and reliability of the WSN application in situations where some sensor is malfunctioning or some radio messages are lost. They are denoted: *ResetMsg*, *ConfirmationMsg* and *LogMsg*.

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 an 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 sensor. The state machine for the experiment control script behavior is illustrated in Figure 3, and consists of the following states:

- *Start* allows users to choose an experiment scenario and establish connection to a serial forwarder software to communicate with the gateway sensor. If the scenario file is loaded and connection to a WSN sensor is established, state *Deploy* is started.

- *Deploy* 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 state *Finish* is started. Otherwise, the coordinates of each sensor are displayed and the script waits while the user places the sensors as described and presses the button. Then, state *Trial* is started.

- *Trial* 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 *Deploy*. Otherwise, state *Sender* is started.

- *Sender* chooses one sensor to play the sender role. The sensor counter is incremented, and if it is equal to the number of sensors in the current deployment, all sensors were given the sender role for the given trial, as the state is changed to *Trial*. Otherwise, measurements are started by state *Reset*.

- *Reset* sends the *ResetMsg* to each sensor to provide synchronization between the sensors. When all sensors are reset, state *Prepare* is started.

- *Prepare* prepares all receivers for beacon reception. It sends *BeaconStartMsg* to every sensor, except the sender node allowing them to be prepared for the reception of beacon messages. When all receivers are ready, state *Measure* is started.

- *Measure* allows the sender to start sending beacons. It sends *BeaconStartMsg* to the sender node 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 received, the next sensor in the scenario is tried for the role of sender. When *BeaconFinishedMsg* is received or the measurement task timeout is expired, state *Transfer* is started.

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

- *Finish* finishes the experiment, and saves collected data.

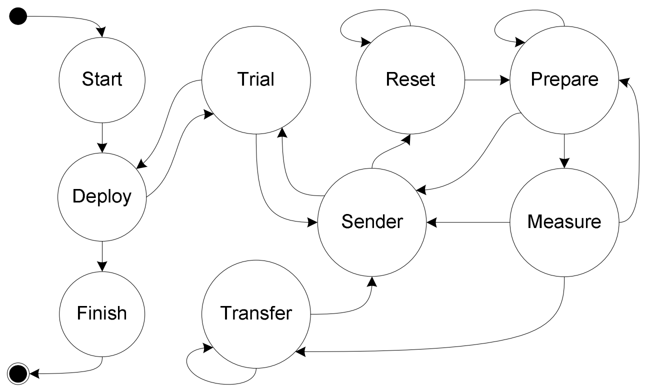

Figure 3. State machine diagram for the experiment control script software behavior

The state machine for the experiment sensor firmware behaviour is illustrated in Figure 4, and consists of the following states:

- *Wait* enables each sensor to wait for the trial start. When the *BeaconStartMsg* message from the host computer is received, the sensor checks the sender field of the message. If the value of this field is equal to the sensor's address, the sensor sets the trial parameters using the values from *beacon_ number*, *tx_power* and *delay* message fields, and changes the state to *Beacon*. Otherwise, it changes the state to *Sample*.

- *Beacon* broadcasts beacons to the receiver nodes. It sets the timer to fire with appropriate frequency, according to the trial parameters. When the timer is expired, the sensor sets the transmission power level to a proper value and sends *BeaconMsg* to the receiver nodes. Also, *StateBeacon* 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 *Wait*. If *ResetMsg* is received during the operation, the measurement is aborted, and the state is changed to *Wait*.

- *Sample* measures RSSI values when beacons from the sender are received. When *BeaconMsg* from the sender is received, the RSSI value is measured and stored into memory. When *TransferStartMsg* from the host computer is received, the state is changed to *Transfer*. If *ResetMsg* is received during sampling, the operation is aborted and the state is changed to *Wait*.

- *Transfer* transfers RSSI samples collected by the receiver 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 *Wait*.

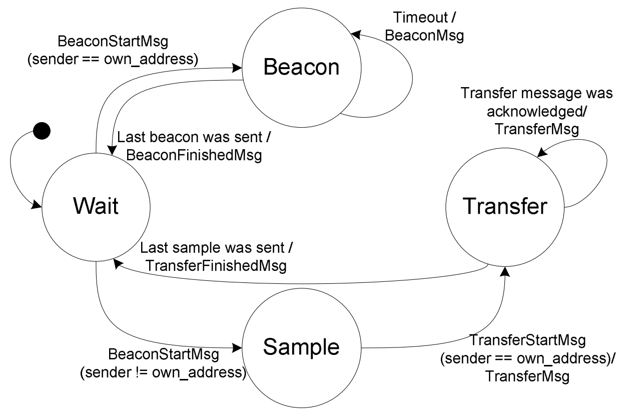
Timeout / BeaconMsg
BeaconStartMsg (sender == own_address)
Beacon
Transfer message was acknowledged/ TransferMsg
Last beacon was sent / BeaconFinishedMsg
Wait
Transfer
Last sample was sent / TransferFinishedMsg
Sample
BeaconStartMsg (sender != own_address)
TransferStartMsg (sender == own_address)/ TransferMsg

Figure 4. State machine diagram for the experiment sensor firmware behavior

3.2. 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 upper 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 of 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 nodes. 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 each 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.

3.3. Implementation of the experiment sensor firmware

The experiment sensor 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 the firmware parameters, and implements 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. 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.

3.4. Implementation of the communication protocol

During running pretests, we have observed that several messages sent by the sensors are getting lost. No component providing reliable information transport was present in the version of TinyOS that 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 the *Automatic Repeat Request* protocol described in [13]. Both experiment control script and sensor 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. Results and Analysis

The goal of the experiments is to determine if all the sensors can be considered equivalent in their abilities to measure RSSI levels. For this purpose, each experiment was containing 4 deployments. Each deployment was using 9 sensors with addresses equal to 9, 1, 2, 3, 4, 5, 6, 7, 8. Sensor 9 was used as a reference sensor. In each deployment, sensors 1, 2, 3, 4, 5, 6, 7, 8 were positioned at the same distance from sensor 9. This distance was equal to 1, 3, 7, 12 meters for each deployment, as shown in Figure 5. During each deployment, reference sensor 9 was sending 15 beacons with transmission power levels equal to 3, 5, 10, 20, 30 respectively.

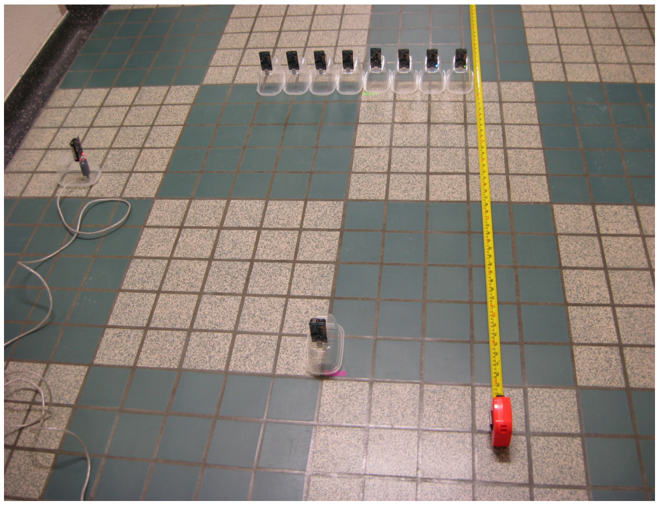

Figure 5. Picture from the experiment deployment

The mean values of RSSI measurements for all devices at different transmission power levels are shown in Figures 6 to 10. 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, the median, the upper quartile 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 verify if the differences in mean RSSI values can be attributed to the differences in the sensors' behavior, analysis of variance (ANOVA) is performed. Such analysis is based on *randomized block design* [14]. 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 j^{th} 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 sensors' behavior). The variation in the dependent variable attributable to such sources is called *nuisance variation*.

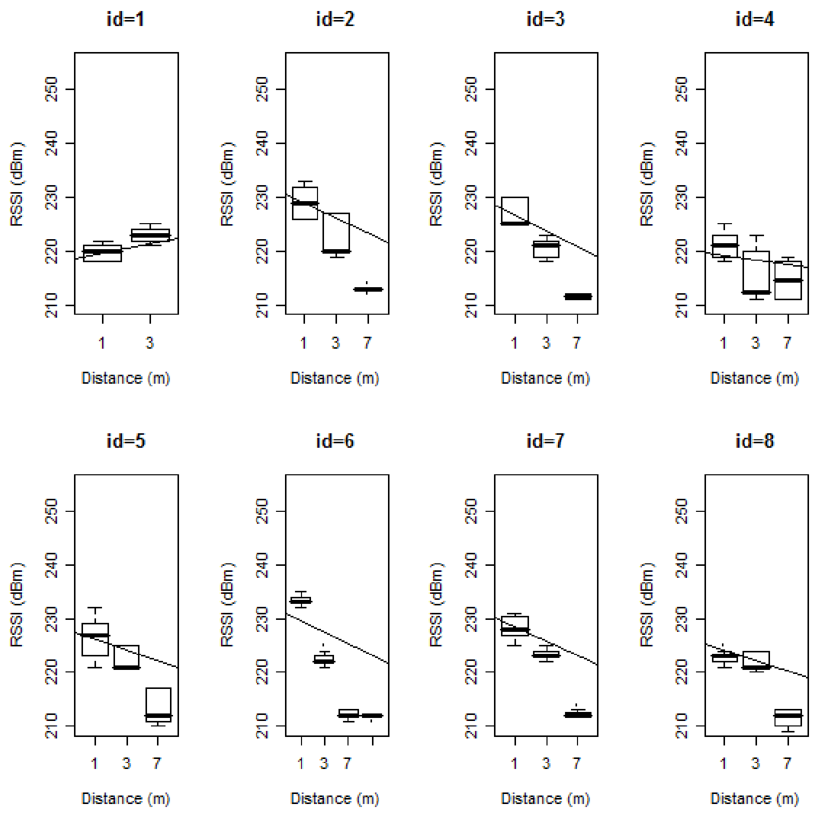

Figure 6. RSSI vs Distance for all sensors for T_X=3

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 levels of nuisance variable. In the given experiments, *n* is equal to the number of sensors and equal to 8, and *m* is the number of different distance levels and is 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 sensor 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 the experiment is called an RB-4 experiment.

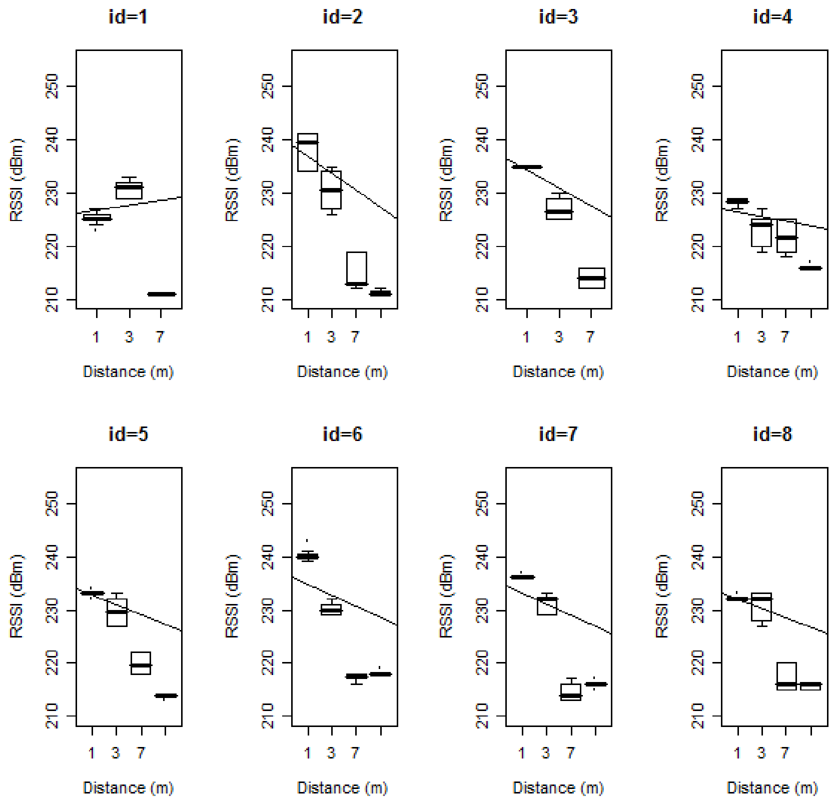

Figure 7. RSSI vs Distance for all sensors for T_X=5

In a randomized block design, a score Y_{ij} is a mean value that reflects the effects of distance level *i*, for sensor *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:

$$Y_{ij} = \mu + \alpha_j + \pi_i + \varepsilon_{ij}, \quad (i = \overline{1,n}, n = 8; j = \overline{1,m},\ m = 4) \qquad (2)$$

where

Y_{ij}	is the mean RSSI value for distance level *j* and sensor *i*;
μ	is the grand mean of the RSSI values $\mu_{11}, \mu_{12}, \ldots, \mu_{nm}$, where μ_{nm} is the RSSI value measured by sensor *n* at distance *m*, with $n = 8$, and $m = 4$;
α_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 $\Sigma^{m}_{j=1}\ \alpha_j = 0$;

π_i is the block effect for particular sensor i, and is equal to μ_i - μ, the deviation of the grand mean from the RSSI mean value for sensor i. The block effect is a normally and independently distributed random variable with mean 0 and standard deviation σ_π^2 (denoted as $NID(0, \sigma_\pi^2)$);

ε_{ij} is the error associated with Y_{ij}, and is equal to Y_{ij} - μ_j - μ_i + μ. The error effect is a random variable that is $NID(0, \sigma_\varepsilon^2)$ and independent of π_i.

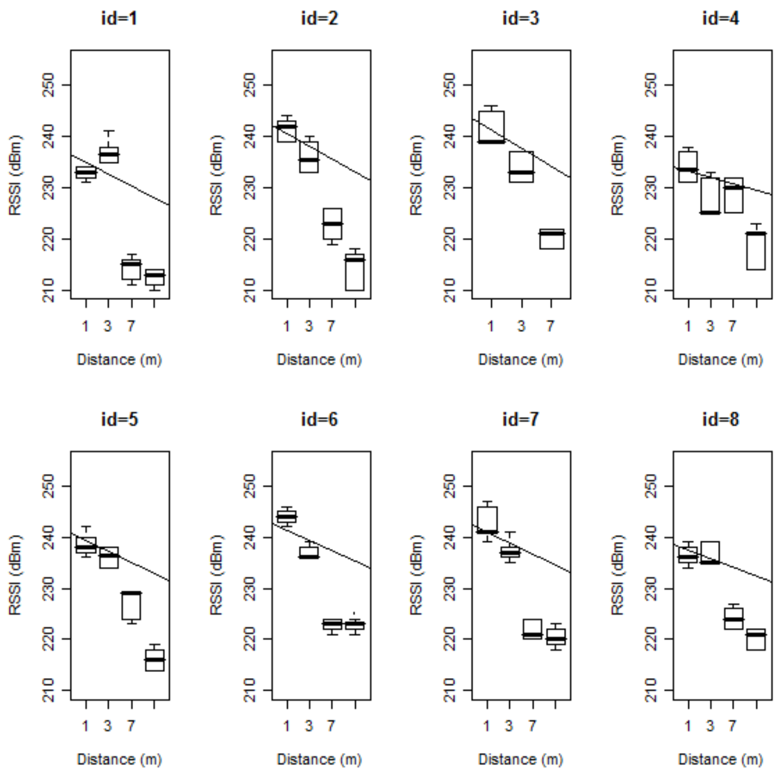

Figure 8. RSSI vs Distance for all sensors for T_X=10

The values of parameters μ, α_j, π_i and ε_{ij} in model (2) are unknown. But, they can be estimated from sample data as follows:

$$
\begin{aligned}
\overline{Y_{ij}} &= \overline{Y} && \text{Grand Mean} \\
&+ (\overline{Y_j} - \overline{Y}) && \text{Effect of the distance} \\
&+ (\overline{Y_i} - \overline{Y}) && \text{Effect of the sensor} \\
&+ (\overline{Y_{ij}} - \overline{Y_j} - \overline{Y_i} + \overline{Y}) && \text{Residual effect}
\end{aligned}
\tag{3}
$$

Equation (3) can be rearranged as follows:

$$\overline{Y_{ij}} - \overline{Y} = (\overline{Y_j} - \overline{Y}) + (\overline{Y_i} - \overline{Y}) + (\overline{Y_{ij}} - \overline{Y_j} - \overline{Y_i} + \overline{Y}) \tag{4}$$

Next, we square both sides of (4) as follows:

$$\left[\overline{Y_{ij}} - \overline{Y}\right]^2 = \left[(\overline{Y_j} - \overline{Y}) + (\overline{Y_i} - \overline{Y}) + (\overline{Y_{ij}} - \overline{Y_j} - \overline{Y_i} + \overline{Y})\right]^2 \tag{5}$$

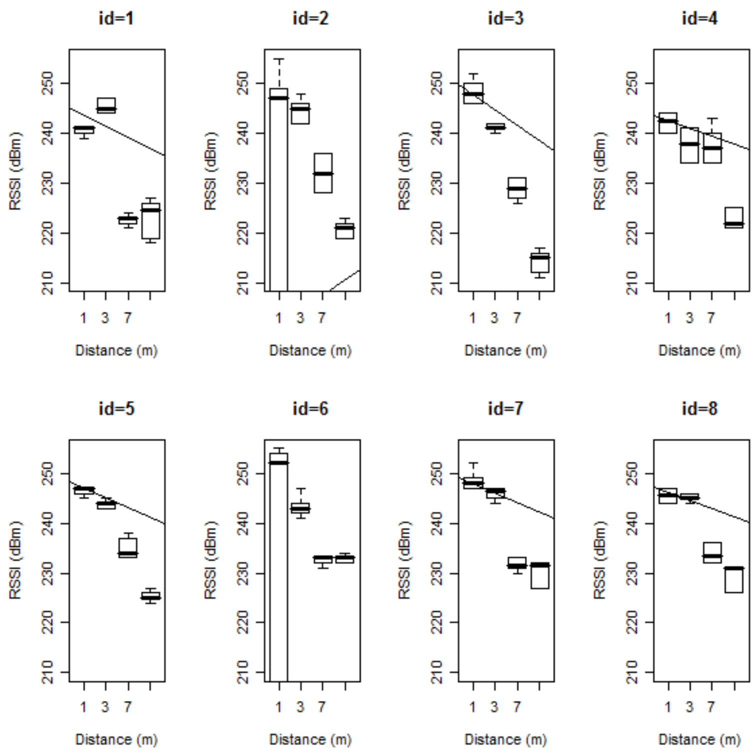

Figure 9. RSSI vs Distance for all sensors for T_X=20

From (5), the total sum of squares (SSTO) could be partitioned into three parts: the sum of squares due to distance A (*SSA*), the sum of squares due to sensors' difference (*SSBL*), and the sum of squares due to residual errors (*SSRES*). The mean squares (*MS*) are obtained by dividing each sum of squares by its degree of freedom [14]. More precisely, we have:

$$MSTO = \frac{SSTO}{nm - 1}$$

$$MSA = \frac{SSA}{m - 1}$$

$$MSBL = \frac{SSBL}{n - 1}$$

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

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

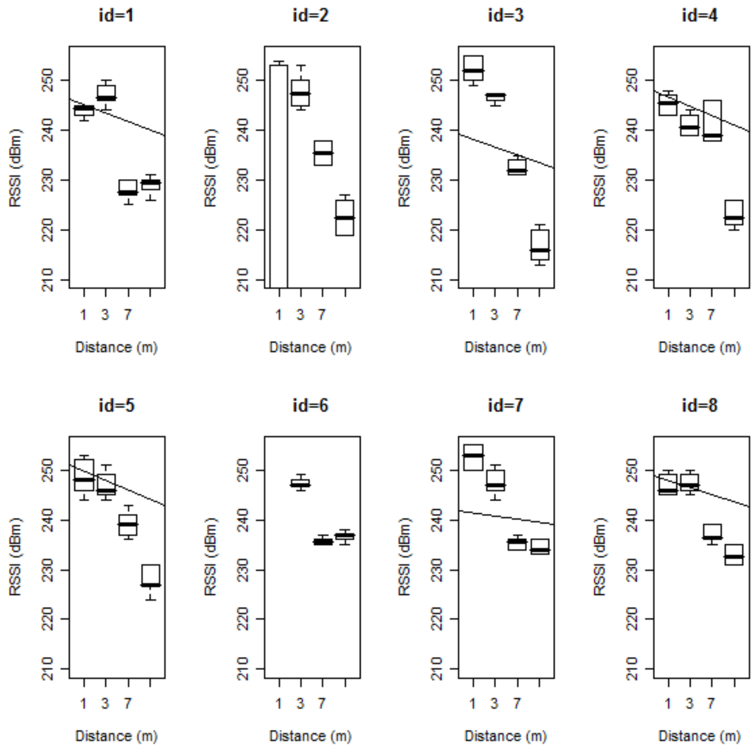

Figure 10. RSSI vs Distance for all sensors for T_X=30

The goal of the analysis is to check two hypotheses H_0 and H_1, first for the distance effect, then for the sensors' effect. The statistical hypotheses for the distance effect are:

$$H_0 : \mu_1 = \mu_2 = \mu_3 = \mu_4 \quad \text{(or } H_0 : \forall j, \alpha_j = 0 \quad \text{for } j \in \overline{1,4}) \tag{6}$$

$$H_1 : \mu_j \neq \mu_{j'} \text{ for } j \neq j' \quad \text{where } j, j' \in \overline{1,4} \quad \text{(or } H_1 : \exists j, \alpha_j \neq 0 \quad \text{for } j \in \overline{1,4}) \tag{7}$$

A test of null hypothesis is given by *FA* = *MSA*/*MSRES* [14]. 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 degree of freedom. It can be expressed as follows:

$$F = \frac{\chi^2_{(d_1)} / d_1}{\chi^2_{(d_2)} / d_2} \tag{8}$$

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 sensor differences are:

$$\begin{aligned} H_0 &: \sigma_\pi^2 = 0 \\ H_1 &: \sigma_\pi^2 \neq 0 \end{aligned} \tag{9}$$

A test of null hypothesis for sensor differences is given by *FBL* = *MSBL*/*MSRES* [14]. 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 sensors 1-8 were receiving beacons from the reference sensor, and the situation where sensors 1-8 were sending beacons to the reference sensor were performed separately. The results for the situation where sensors 1-8 was receiving beacons from reference sensor 9 are presented in Table 1, whereas the analysis results for the situation where sensors 1-8 were sending beacons from reference sensor 9 are presented in Table 2. 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.

Table 1. Variation of RSSI due to the sensors difference (reception)

Source	***df***	***SS***	***MS***	***F***	***p***
Receiver	7	484.29	69.18	0.55	0.79
Distance	3	585.83	195.28	1.556	0.23
Residuals	21	2635.44	125.50		

Table 2. Variation of RSSI due to the sensors difference (transmission)

Source	***df***	***SS***	***MS***	***F***	***p***
Sender	7	2535.83	362.26	0.96	0.48
Distance	3	1814.2	604.7	1.5955	0.22
Residuals	21	7959.3	379.0		

For the situation where sensors 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 sensors' 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 sensors 1-8 were sending beacons, we obtained: $FBL = 0.96$, $p = 0.48 > 0.05$. It means that the differences in RSSI values due to the difference in sensors are non significant. Similarly, we obtained: $FA = 1.60$, $p = 0.22$. 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 sensors at all transmission power levels together, as the difference at RSSI recorded for different transmission power masks the effect of the changing distance. The results allow to conclude that the sensors are manufactured with enough precision, so that no calibration is needed to compensate differences in the sensors' behavior.

5. CONCLUSION

In this paper, we presented an experiment for evaluating the sensors' behavior in wireless sensor networks (WSN). More specifically, a WSN application was implemented for collecting RSSI measurements in different conditions. The application consists of two parts: an experiment control script which runs on a computer, and an experiment sensor firmware which runs on each WSN node. The experiment control script sends experiment configuration parameters to each sensor, commands each sensor to start measurements accordingly to experiment scenarios, and collects measured data. The experiment sensor firmware performs RSSI measurements.

Statistical analysis of variance (ANOVA) was performed to determine the factors affecting the RSSI measurements. Results analysis shows that the sensors are manufactured with enough precision, so that no calibration is needed to compensate differences in the sensors' behavior. Future work should be oriented towards implementing the positioning methods which take into account the variability in RSSI measurements.

REFERENCES

[1] N. Wang, N. Zhang, and M. Wang, "Wireless sensors in agriculture and food industry recent development and future perspective," Computers and Electronics in Agriculture, vol. Volume 50, Issue 1, pp. 1 – 14, January 2006.

[2] M. Kuorilehto, M. H¨annik¨ainen, and T. D. H¨am¨al¨ainen, "A survey of application distribution in wireless sensor networks," EURASIP J. Wirel. Commun. Netw., vol. 5, no. 5, pp. 774–788, 2005.

[3] K. Romer and F. Mattern, "The design space of wireless sensor networks," Wireless Communications, IEEE [see also IEEE Personal Communications], vol. 11, no. 6, pp. 54–61, Dec. 2004.

[4] A. Savvides, C.-C. Han, and M. B. Srivastava, "Dynamic fine-grained localization in ad-hoc networks of sensors," in MOBICOM, 2001, pp. 166–179.

[5] N. Patwari, J. Ash, S. Kyperountas, I. Hero, A.O., R. Moses, and N. Correal, "Locating the nodes: cooperative localization in wireless sensor networks," Signal Processing Magazine, IEEE, vol. 22, no. 4, pp. 54–69, July 2005.

[6] C. Alippi and G. Vanini, “A RSSI-based and calibrated centralized localization technique for Wireless Sensor Networks,” in Pervasive Computing and Communications Workshops, 2006. PerCom Workshops 2006. Fourth Annual IEEE International Conference on, 13-17 March 2006, pp. 301–305.

[7] X. Li, H. Shi, and Y. Shang, “A sorted RSSI quantization based algorithm for sensor network localization,” in Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on, vol. 1, 20-22 July 2005, pp. 557–563Vol.1.

[8] H. Chen, D. Ping, Y. Xu, and X. Li, “A novel localization scheme based on RSS data for Wireless Sensor Networks,” in APWeb Workshops, ser. Lecture Notes in Computer Science, H. T. Shen, J. Li, M. Li, J. Ni, and W. Wang, Eds., vol. 3842. Springer, 2006, pp. 315–320.

[9] Moteiv, “Tmote Sky ultra low power IEEE 802.15.4 compliant wireless sensor module datasheet,” 2006.

[10] Chipcon, “CC2400 2.4 GHz low-power RF transceiver datasheet,” Texas Instruments, 2004.

[11] “TinyOS: An open-source OS for the wireless embedded sensor networks,” http://www.tinyos.net. [Online]. Available: http://www.tinyos.net

[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesC language: A holistic approach to networked embedded systems,” ACM SIGPLAN Notices, vol. 38, no. 5, pp. 1–11, May 2003.

[13] A. S. Tanenbaum, Computer networks, 4th ed. Prentice Hall, March 17 2003.

[14] R. E. Kirk, Experimental design: procedures for the behavioral sciences, 3rd ed. Brooks/Cole Publishing Company, 1995.