Node Localization in Wireless Ad-Hoc Sensor Networks

(February-June 2005)

Submitted in the partial fulfillment of requirements for VIII Semester, B.E.,
Computer Science and Engineering

Under
Visveswariah Technological University, Belgaum
by

Nalini Vasudevan (1RV01CS059)

under the guidance of

Internal Guide	External Guide
Mrs. Shobha.G	**Dr. Anurag Kumar**
Assistant Professor,	Prof. and Chairman,
CSE Dept, RVCE	ECE Dept, IISc

Department of Computer Science and Engineering
R.V. College of Engineering, Bangalore

R.V. COLLEGE OF ENGINEERING
Department of Computer Science and Engineering
Bangalore - 59

CERTIFICATE

This is to certify the project titled **Node Localization in Wireless Ad-Hoc Sensor Networks** has been successfully completed by

Nalini Vasudevan (1RV01CS059)

in partial fulfillment of VIII Semester B.E, (Computer Science and Engineering), during the period March-June 2005 as prescribed by the Visveswariah Technological University, Belgaum. **The project developed is an original one and completed during the semester course work.**

Nalini Vasudevan

Signature of

Guide, Dept. of CSE **HOD, Dept. of CSE** **Principal, RVCE**

Examiner 1 **Examiner 2**

TO WHOM IT MAY CONCERN

This is to certify that the following student:

Nalini Vasudevan

has satisfactorily completed their project **Node Localization of Wireless Ad-Hoc Sensor Networks** during the period February to May 2005, under my guidance.

Dr.Anurag Kumar,
Professor and Chairman,
Electrical Communication Engg. Dept.,
Indian Institute of Science (IISc),
Bangalore-5600012, INDIA.

Acknowledgment

A project is never a solo effort. For those who never peeked behind the scenes, discovering just how many people are involved in a project is a real eye-opener. This is where we take the opportunity to thank each of them. I feel greatly privileged to express thanks to all the people who helped us to complete the project successfully.

I would like to thank **Dr. Anurag Kumar**, Chairman and Professor of Electrical and Communication Engineering Department, Indian Institute of Science, for giving us the opportunity and infrastructure to work on this project and for providing his expertise and guidance throughout the duration of the project.
A special word of thanks to the technical staff of the Communication lab at IISc, specially **Manjunath D** for extending their support and expertise to us.
I would like to acknowledge **Anushruthi Rai**: we collaborated together on the same project.
I convey our sincere regards to our internal guide **Mrs Shobha**, Assistant Professor, RV college of engineering, for providing invaluable suggestions and guidance at all stages of the project.
I wish to place on record, our grateful thanks to **Prof. B.I.Kodhanpur**, head of department of computer science, RVCE for providing us encouragement and guidance throughout our work.

Contents

1 Synopsis **1**

2 Introduction **2**

2.1 Background 2

2.1.1 Sensor Sub-systems 3

2.1.1.1 A computing subsystem 3

2.1.1.2 A communication subsystem 3

2.1.1.3 A sensing subsystem 3

2.1.1.4 A power supply subsystem 3

2.1.2 Challenges 3

2.1.3 Important aspects 4

2.1.3.1 Localization 4

2.1.3.2 Energy Efficiency 4

2.1.3.3 Routing 5

2.1.3.4 Media Access Control in Sensor Networks 5

2.1.4 Sensor networks applications 7

2.1.5 Simulators for Sensor Networks 8

2.2 Localization 9

2.2.1 Localization Techniques 9

3 System Requirement Specification **12**

3.1 Hardware specifications 12

3.1.1 Crossbow Mica mote and sensors 12

3.2 Software Specifications 14

3.2.1 TinyOS 14

3.2.1.1 TinyOS Application 14

3.2.2 nesC 15

3.2.3 TinyDB 16

3.2.3.1 System Overview 17

3.2.3.2 Installation and Requirements 18

3.2.3.3 Running the TinyDBMain GUI 18

3.2.3.4 Using TinyDB 19

3.2.3.5 The TinyDB Java API 20

3.2.3.6 The TinyDB Demo Application 21

3.2.3.7 TinyDB Source Files 22

3.3 Functional Requirements 24

3.3.1 Non Functional Requirements 25

3.3.2 Scalability 25

3.3.3 Efficient Data Propagation 25

3.3.4 Memory Efficiency 25

4 Design 26

4.1 Data Flow Diagram 26

4.2 Basic Design 26

4.3 Algorithm 28

5 Testing and Results 29

5.1 White Box Testing 29

5.1.1 Unit Testing 29

5.1.1.1 TinyOS Installation Verification 29

5.1.1.2 Hardware verification 30

5.1.1.3 Radio Verification 31

5.1.1.4 TinyDB Verification 32

5.1.1.5 GetNeighbours module 32

5.1.1.6 Localization module 33

5.1.2 Integrated Testing 33

5.1.3 Results : Black Box Testing 35

5.1.3.1 Example 1: A mote in the neighborhood of one beacon 35

5.1.3.2 Example 2: A mote in the neighborhood of two beacons 35

5.1.3.3 Example 3: A mote in the neighbourhood of one beacon but not the other 39

5.1.3.4 Example 4: A mote in the neighbourhood of no beacons 41

6 Conclusion 43

7 Limitations and Future Enhancement 44

A nesC Files : Detection of Neighbors 45

A.1 Query for Neighbors 45

A.1.1 SendQueryC.nc 45

A.1.1.1 SendQueryM.nc 45

A.2 Respond to Request/Neighbor Information 46

A.2.1 Receiver.h 46

A.2.2 ReceiverC.nc 47

A.2.3 ReceiverM.nc 47

B Java File : Localization.java 52

List of Figures

2.1 A typical sensor network 2
2.2 Beacon Mote and Localization 10

3.1 Crossbow's mica2 mote and sensor 13

4.1 0-Level DFD 26
4.2 1-Level DFD 27
4.3 Localization with IR 27

5.1 TinyDB GUI 33
5.2 Modified frontend to encorporate localization. 34
5.3 Sensor Readings 35
5.4 Localization: A multihop scenario 36
5.5 Example 1: The actual testbed 37
5.6 Example 1: Output 37
5.7 Example 2: The actual testbed 38
5.8 Example 2: Output 38
5.9 Example 3: The actual testbed 40
5.10 Example 3: Output 40
5.11 Example 4: The actual testbed 42
5.12 Example 4: Output 42

Chapter 1

Synopsis

Localization is a fundamental task in wireless ad-hoc networks. We consider the problem of locating and orienting a network of unattended sensor nodes that have been deployed in a scene at unknown locations. In a location related system, the aquisition of objects' locations is the critical step for the effective and smooth working procedures.The basic concept is to deploy a large number of low-cost, self-powered sensor nodes that acquire and process data. The sensor nodes may include one or more acoustic microphones as well as seismic, magnetic, or imaging sensors. A typical sensor network objective is to detect, track, and classify objects or events in the neighborhood of the network.

We consider location estimation in networks where a small proportion of devices, called reference devices or beacons, have a priori information about their coordinates. All devices,regardless of their absolute coordinate knowledge, estimate the range between themselves and their neighboring devices. Such location estimation is called relative location because the range estimates collected are predominantly between pairs of devices of which neither has absolute coordinate knowledge.
We intend to implement a simple method of localization called the in-range method IR to a two-dimensional network of sensor nodes. The basic premise of IR is that the transmission at a given power can be decoded only upto a maximum distance called its transmission range. Therefore, if a node is able to receive a signal from a beacon, this would imply that the node can be localized to a set of positions represented by a disc of radius equal to the transmission range. Similarly a localized node would be able to aid the localization of its other neighbours. In this way, an iterative process could be used, by which the sensors can collaboratively learn and improve their localization regions .
Nodes use this simple connectivity metric, to infer proximity to a given subset of these reference points. Nodes localize themselves to their proximate reference points. The accuracy of localization is then dependent on the separation distance between two adjacent reference points and the transmission range of these reference points.

Chapter 2

Introduction

Sensor networks are dense wireless networks of small, low-cost sensors, which collect and disseminate environmental data. Wireless sensor networks facilitate monitoring and controlling of physical environments from remote locations with better accuracy. They have applications in a variety of fields such as environmental monitoring, military purposes and gathering sensing information in inhospitable locations.

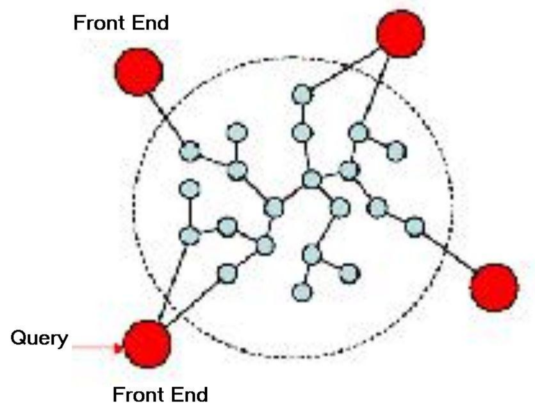


Figure 2.1: A typical sensor network

Sensor nodes have various energy and computational constraints because of their inexpensive nature and ad-hoc method of deployment. Considerable research has been focused at overcoming these deficiencies through more energy efficient routing, localization algorithms and system design. Our survey attempts to provide an overview of these issues as well as the solutions proposed in recent research literature.

2.1 Background

Previously, sensor networks consisted of small number of sensor nodes that were wired to a central processing station. However, nowadays, the focus is more on wireless, distributed, sensing nodes. When the exact location of a particular phenomenon is unknown, distributed sensing allows for closer placement to the phenomenon than a single sensor would permit[9]. Also, in many cases, multiple sensor nodes are required to overcome environmental obstacles like obstructions, line of sight constraints etc. In most cases, the environment to be monitored does not have an existing infrastructure for either energy or communication. It becomes imperative for sensor nodes to survive on small, finite sources of energy and communicate through a wireless communication channel. Another requirement for sensor networks would be distributed processing capability. This is necessary since communication is a major consumer of energy. A centralized system would mean that some of the sensors would need to communicate over long distances that lead to even more energy depletion. Hence, it would be a good idea to process locally as much information as possible in order to minimize the total number of bits transmitted.

2.1.1 Sensor Sub-systems

A sensor node usually consists of four sub-systems[10]:

2.1.1.1 A computing subsystem

It consists of a microprocessor(micro controller unit, MCU), which is responsible for the control of the sensors and execution of communication protocols. MCUs usually operate under various operating modes for power management purposes. But shuttling between these operating modes involves consumption of power, so the energy consumption levels of the various modes should be considered while looking at the battery lifetime of each node.

2.1.1.2 A communication subsystem

It consists of a short-range radio, which is used to communicate with neighboring nodes and the outside world. Radios can operate under the Transmit, Receive, Idle and Sleep modes. It is important to completely shut down the radio rather than put it in the idle mode when it is not transmitting or receiving because of the high power consumed in this mode.

2.1.1.3 A sensing subsystem

It consists of a group of sensors and actuators and links the node to the outside world. Using low power components and saving power at the cost of performance, which is not required, can reduce energy consumption.

2.1.1.4 A power supply subsystem

It consists of a battery, which supplies power to the node. It should be seen that the amount of power drawn from a battery is checked because if high current is drawn from a battery for a long time, the battery will die even though it could have gone on for a longer time. Usually the rated current capacity of a battery being used for a sensor node is lesser than the minimum energy consumption required leading to the lower battery lifetimes. Reducing the current drastically or even turning it off often can increase the lifetime of a battery.

2.1.2 Challenges

In spite of the diverse applications, sensor networks pose a number of unique technical challenges due to the following factors:

- **Ad hoc deployment**: Most sensor nodes are deployed in regions, which have no infrastructure at all. A typical way of deployment in a forest would be tossing the sensor nodes from an aeroplane. In such a situation, it is up to the nodes to identify its connectivity and distribution.
- **Unattended operation**: In most cases, once deployed, sensor networks have no human intervention. Hence the nodes themselves are responsible for reconfiguration in case of any changes.
- **Untethered**: The sensor nodes are not connected to any energy source. There is only a finite source of energy, which must be optimally used for processing and communication. An interesting fact is that communication dominates processing in energy consumption. Thus, in order to make optimal use of energy, communication should be minimized as much as possible.
- **Dynamic changes**: It is required that a sensor network system be adaptable to changing connectivity (for e.g., due to addition of more nodes, failure of nodes etc.) as well as changing environmental stimuli.

Thus, unlike traditional networks, where the focus is on maximizing channel throughput or minimizing node deployment, the major consideration in a sensor network is to extend the system lifetime as well as the system robustness [8].

2.1.3 Important aspects

2.1.3.1 Localization

In most of the cases, sensor nodes are deployed in an ad hoc manner. It is up to the nodes to identify themselves in some spatial co-ordinate system. This problem is referred to as localization. This aspect is discussed later.

2.1.3.2 Energy Efficiency

Energy consumption is the most important factor to determine the life of a sensor network because usually sensor nodes are driven by battery and have very low energy resources. This makes energy optimization more complicated in sensor networks because it involved not only reduction of energy consumption but also prolonging the life of the network as much as possible. Having energy awareness in every aspect of design and operation can do this. This ensures that energy awareness is also incorporated into groups of communicating sensor nodes and the entire network and not only in the individual nodes.
Developing design methodologies and architectures, which help in energy aware design of sensor networks, can reduce the power consumed by the sensor nodes. The lifetime of a sensor network can be increased significantly if the operating system, the application layer and the network protocols are designed to be energy aware. Power management in radios is very important because radio communication consumes a lot of energy during operation of the system. Another aspect of sensor nodes is that a sensor node also acts a router and a majority of the packets, which the sensor receives, are meant to be forwarded. Intelligent radio hardware that help in identifying and redirecting packets which need to be forwarded and in the process reduce the computing overhead because the packets are no longer processed in the intermediate nodes.

Traffic can also be distributed in such a way as to maximize the life of the network. A path should not be used continuously to forward packets regardless of how much energy is saved because this depletes the energy of the nodes on this path and there is a breach in the connectivity of the network. It is better that the load of the traffic be distributed more uniformly throughout the network. It is important that the users be updated on the health of a sensor network because this would serve as a warning of a failure and aid in the deployment of additional sensors.

2.1.3.3 Routing

Conventional routing protocols have several limitations when being used in sensor networks due to the energy-constrained nature of these networks. These protocols essentially follow the flooding technique in which a node stores the data item it receives and then sends copies of the data item to all its neighbors.

- **Implosion**: If a node is a common neighbor to nodes holding the same data item, then it will get multiple copies of the same data item. Therefore, the protocol wastes resources sending the data item and receiving it.
- **Resource management**: In conventional flooding, nodes are not resource-aware. They continue with their activities regardless of the energy available to them at a given time.

The routing protocols designed for sensor networks should be able to overcome both these deficiencies or/and look at newer ways of conserving energy increasing the life of the network in the process. Ad-hoc routing protocols are also unsuitable for sensor networks because they try to eliminate the high cost of table updates when there is highly mobility of nodes in the network. But unlike ad-hoc networks, sensor networks are not highly mobile. Routing protocols can be divided into proactive and reactive protocols. Proactive protocols attempt at maintaining consistent updated routing information between all the nodes by maintaining one or more routing tables. In reactive protocols, the routes are only created when they are needed. The routing can be either source-initiated or destination-initiated.

2.1.3.4 Media Access Control in Sensor Networks

Media Access Control in sensor networks is very different than in the traditional networks because of its constraints on computational ability, storage and energy resources. Therefore media access control should be energy efficient and should also allocate bandwidth fairly to the infrastructure of all nodes in the network. In sensor networks, the primary objective is to sample the residing environment for information and send it to a higher processing infrastructure (base station) after processing it. The data traffic may be low for lengthy periods with intense traffic in between for short periods of time. Most of the time, the traffic is multihop and heading towards some larger processing infrastructure. At each of the nodes, there is traffic originating out of the node and traffic being routed through the node because most nodes are both data sources and routers. There are several limitations on sensor nodes too. They have little or no dedicated carrier sensing or collision detection and they have no specific protocol stacks, which could specify the design of their media access protocol.

- **Fairness**

  The following are the challenges in multihop sensor networks

- The originating traffic from a node has to compete with the traffic being routed through that node.
- An undetected node might exist in the network, which might result in unexpected contention for bandwidth with route-thru traffic.
- The probability of corruption and contention at every hop is higher for the nodes, which reside farther away from the higher processing infrastructure.
- Energy is invested in every packet when it is routed through every node. Therefore, the longer a packet has been routed, the more expensive it is to drop that packet.

Listening to the network, which is expensive, can do carrier sensing in sensor networks. Therefore the listening period should be short to conserve energy. The traffic also tends to be highly synchronized because nearby nodes tend to send messages to report the same event. Since there is no collision detection, the nodes will tend to corrupt each other's messages when they send them at the same time. This could happen every time they detect a common event. To reduce contentions, a back off mechanism could be used. A node could restrain itself from transmitting for a certain period of time and hopefully the channel becomes clear after the back off period. This will help in desynchronizing the traffic too. Contention protocols in traditional networks widely use the Request to Send(RTS), Clear to Send(CTS) and acknowledgements(ACK) to reduce contentions. A RTS-CTS-DATA-ACK handshake is extremely costly though when used in sensor networks because every message transmitted uses up the low energy resources of the nodes. Therefore, the number of control packets used should be kept as low as possible. Thus, only the RTS and CTS messages are used in the control scheme. If a node does not receive the CTS after sending the RTS for a long time, the node will back off for a binary exponentially increasing time period and then transmit again. If it receives a CTS, which is not meant for it or receives a CTS before its own transmission, it will back off to avoid collisions. Fairness in allocation between the originating traffic and route-thru traffic should be achieved. The media access controls the originating traffic when the route-thru traffic is high and when the originating traffic is high, it applies a backpressure to control the route-thru traffic deep down in the network from where it originated. A linear increase and multiplicative decrease approach is used for transmission control. The transmission rate control is probabilistic and it is linearly increased by a constant and it is decreased by multiplying it with, a, where a is less than 1 and greater than 0. Since dropping traffic which is being routed through is wastage of the network's energy resources, more preference is given to it by making its dropping penalty 50

The advantage of this scheme is that the amount of computation required for this is within the sensor nodes' computational capability and achieves good energy efficiency when the traffic is low while maintaining the fairness among the nodes.

- **S-MAC** .The major sources of energy wastage are:
  - Collisions
  - Overhearing
  - Control packet overhead

– Idle listening

Unlike in traditional networks where all nodes require equal opportunity to transmit, sensor nodes all try to achieve a single common task. S-MAC introduces uses three techniques to reduce energy consumption. Firstly the nodes go to sleep periodically so that they do not waste energy by listening to an empty channel or when a neighboring node is transmitting to another node. This helps in avoiding the overhearing problem too. Secondly, nearby nodes form virtual clusters to synchronize their wake-up and sleep periods to keep the control packet overhead of the network low. Finally, message passing is used to reduce the contention latency and control overhead. S-MAC consists of three components

– **Periodic Listen and Sleep**: Neighboring nodes are synchronized to go to sleep together so as to avoid a heavy control overhead. They listen together and sleep together. For this the nodes exchange schedules with their immediate neighbors. The nodes use RTS and CTS to talk to each other and contend for the medium if they want to communicate with the same node. Synchronized nodes form a virtual cluster but there is no real clustering and no inter-cluster communication problem. Synchronization is maintained by using SYNC packets, which contain the sender's address and its next sleep time.

– **Collision and Overhearing Avoidance**: S-MAC adopts a contention-based scheme to avoid collisions. A duration field is introduced in each transmitted packet, which indicates how much longer the transmission will last. When a node receives a packet, it will not transmit any packets for at least the time that is specified in the duration field. This is recorded in a variable in the node called the Network Allocation Vector (NAV), which is reset every time the node received a packet whose duration field is larger than the current value. When the NAV is zero, the node can start transmitting packets. Overhearing is avoided by letting the nodes, which get RTS and CTS packets, which are not meant for them, go to sleep. All immediate neighbors also go to sleep till the current transmission is completed after a sender or receiver receives the RTS or CTS packet.

– **Message Passing**: Long messages are fragmented into smaller messages and transmitted in a burst. This is to avoid the high overhead and delay encountered for retransmitting when a long message is lost. ACK messages are used to indicate if a fragment is lost at any time so that the sender can resend the fragment again. The ACK messages also have the duration field to reduce overhearing and collisions. There is no contention to achieve fairness for each lost fragment. It is allowed to retransmit the current fragment but there is a limit on the number of retransmissions the node is allowed without any contention.

2.1.4 Sensor networks applications

Sensor networks may consist of many different types of sensors as discussed in [2] such as seismic, low sampling rate magnetic, thermal, visual, infrared, acoustic and radar, which are able to monitor wide variety of ambient conditions that include the following :

- temperature
- humidity
- vehicular movement
- lightning condition
- pressure
- soil makeup
- noise levels
- the presence or absence of certain kinds of objects
- mechanical stress levels on attached objects
- the current characteristics such as speed, direction and size of an object.

Sensor nodes can be used for continuous sensing, event detection, event ID, location sensing, and local control of actuators. The concept of micro-sensing and wireless connection of these nodes promise many new application areas. We categorize the applications into military, environment, health, home and other commercial areas. It is possible to expand this classification with more categories such as space exploration, chemical processing and disaster relief. In fact, due to the pervasive nature of micro-sensors, sensor networks have the potential to revolutionize the very way we understand and construct complex physical system[7].

2.1.5 Simulators for Sensor Networks

For the sake of completeness, this section very briefly looks at some of the more prominent simulators for sensor networks available today:

- **GloMoSim [11]**: GLobal Mobile Information systems Simulator are a scalable simulation environment for wireless and wired network systems. It is written both in C and Parsec. It is capable of parallel discrete-event simulation. GloMoSim currently supports protocols for a purely wireless network. A basic level of Parsec knowledge and thorough C knowledge is sufficient to carry out simulations
- **NS-2 [15]**: The mother of all network simulators has facilities for carrying out both wireless and wired simulations. It is written in C++ and oTCL. Since it is object-oriented, it is easier to add new modules. It provides for support for energy models. Some example applications are included as a part of the package. It has the advantage of extensive documentation.
- **SensorSim [17]**: is a simulation framework for sensor networks. It is an extension to the NS simulator. It provides the following: Sensing channel and sensor models, Battery models, Lightweight protocol stacks for wireless micro sensors, Scenario generation and Hybrid simulation. It is geared very specifically towards sensor networks and is still in the pre-release stage. It does not have proper documentation.

2.2 Localization

In sensor networks, nodes are deployed into an unplanned infrastructure where there is no a priori knowledge of location. The problem of estimating spatial-coordinates of the node is referred to as localization. An immediate solution, which comes to mind, is GPS or the Global Positioning System. The different approaches to the localization problem have been studied in [3, 4, 5]. However, there are some strong factors against the usage of GPS. For one, GPS can work only outdoors. Secondly, GPS receivers are expensive and not suitable in the construction of small cheap sensor nodes. A third factor is that it cannot work in the presence of any obstruction like dense foliage etc. Thus, sensor nodes would need to have other means of establishing their positions and organizing themselves into a co-ordinate system without relying on an existing infrastructure. Most of the proposed localization techniques today, depend on recursive trilateration/multilateration techniques[8]. One way of considering sensor networks is taking the network to be organized as a hierarchy with the nodes in the upper level being more complex and already knowing their location through some technique (say, through GPS). These nodes then act as beacons by transmitting their position periodically. The nodes, which have not yet inferred their position, listen to broadcasts from these beacons and use the information from beacons with low message loss to calculate its own position. A simple technique would be to calculate its position as the centroid of all the locations it has obtained. This is called as proximity based localization. It is quite possible that all nodes do not have access to the beacons. In this case, the nodes, which have obtained their position through proximity, based localization themselves act as beacons to the other nodes. This process is called iterative multilateration. As can be guessed, iterative multilateration leads to accumulation of localization error. Thus, trilateration is a geometric principle which allows us to find a location if its distance from three already-known locations. The same principle is extended to three-dimensional space. In this case, spheres instead of circles are used and four spheres would be needed. When a localization technique using beacons is used, an important question would be 'how many initial beacons to deploy. Too many beacons would result in self-interference among the beacons while too less number of beacons would mean that many of the nodes would have to depend on iterative multilateration.

2.2.1 Localization Techniques

Localization can be classified as fine-grained, which refers to the methods based on timing/signal strength and coarse-grained, which refers to the techniques based on proximity to a reference point. [6] gives an over-view of the various localization techniques. Examples of fine-grained localization are:

- **Timing**: The distance between the receiver node and a reference point is determined by the time of flight of the communication signal.
- **Signal strength**: As a signal propagates, attenuation takes place proportional to the distance traveled. This fact is made use of to calculate the distance.
- **Signal pattern matching**: In this method, the coverage area is pre-scanned with transmitting signals. A central system assigns a unique signature for each square in the location grid. The system matches a transmitting signal from a mobile transmitter with the pre-constructed database and arrives at the correct

location. But pre-generating the database goes against the idea of ad hoc deployment.

- **Directionality**: Here, the angle of each reference point with respect to the mobile node in some reference frame is used to determine the location.

Examples of coarse-grained localization is proximity based localization as described earlier. [6] proposes a localization system which is RF-based, receiver-based, ad hoc, responsive, low-energy consuming and adaptive. RF-based transceivers would be more inexpensive and smaller compared to GPS-receivers. Also in an infrastructure less environment, the deployment would be ad hoc and the nodes should be able to adapt themselves to available reference points.

Locating objects in two (e.g., surface of the earth) or three dimensions (e.g., space) from the knowledge of locations of some distinguished nodes, called beacons, has been the central problem in navigation. Beacons can know location of a node from its distances and/or angles. What distinguishes the localization problem in sensor networks from the navigation problem is the following. Due to spatial expanse of a sensor network not every sensor will have the required number of beacons for ranging; to be cost effective, fewer beacons are desired.

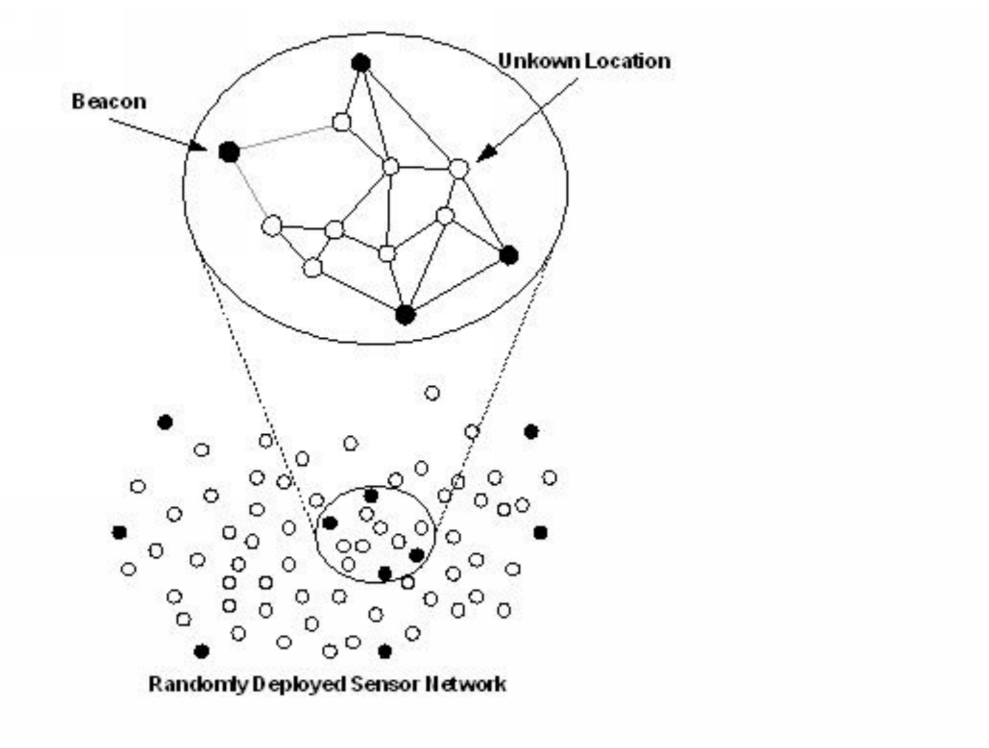


Figure 2.2: Beacon Mote and Localization

In addition, the traditional ranging methods based on received signal strength (RSSI), time of arrival (TOA), angle of arrival (AOA), time difference of arrival (TDOA), etc. have several shortcomings from the point of view of the sensor networks. RSSI is usually very unpredictable since the received signal power is a complex function of the propagation environment. Hence, radios in sensors will need to be well calibrated otherwise sensors may exhibit significant variation in power to distance mapping. TOA using acoustic ranging will require an additional ultrasound source. TOA and RSSI are affected by measurement as well as non-line of sight errors. TDOA is not very

practical for a distributed implementation. AOA sensing will require either an antenna array or several ultra-sound receivers. This motivates us to consider a particularly simple method of localization as in [1], which we call the in-range method (IR). Here we implement a simple method of localization in sensor networks in which a sensor with unknown location is localized to a disk of radius equal to the transmission range centered at a beacon if the sensor under consideration can receive a transmission from the beacon. This is a reliable and extremely easy-to-implement technique since it assumes only a basic communication capability. The real advantage, however, is that once localized, a sensor aids the other sensors in localization. This way by collaboration sensors can learn and improve their localization regions iteratively. We analyze this iterative scheme and construct a distributed algorithm for utilizing it in real sensor networks. The basic premise of IR is that a transmission at a given power can be decoded only up to a maximum distance, called its transmission range. IR then simply localizes a node with unknown location to a disk of radius equal to the range centered at a beacon if the node under consideration can successfully decode a transmission from the beacon.
.

Chapter 3

System Requirement Specification

3.1 Hardware specifications

The hardware required is as follows:

- **Motes** (having a processor with small memory)
- **Sensors** (to be attached to the motes, to detect temperature, light etc).
- **Antennas** (for wireless communication between the motes)
- **Base Station** (attached to the serial port of the PC/Laptop)
- **PC/Laptop** (to collect the readings from the motes and display the readings and positions)

3.1.1 Crossbow Mica mote and sensors

Crossbows [16] wireless sensor platform gives the flexibility to create powerful, tether-less, and automated data collection and monitoring systems. Crossbows supports a wide range of hardware and sensors for various customer requirements. Most of the hardware can plug-and-play and it all runs TinyOS / nesC from UC Berkeley. The platform consists of Processor Radio boards (MPR) commonly referred to as MOTES. These battery powered devices run TinyOS and support two-way mesh radio networks. Sensor and data acquisition cards (MTS and MDA) plug into the Mote Processor Radio boards. Sensor support includes both direct sensing as well interfaces for external sensors. Finally, gateway and interface products (MIB), allow customers to interface Motes to PCs, PDAs, the WWW, and existing wired networks and protocols. The TinyOS operating system is open-source, extendable, and scalable. Code modules are wired together allowing fluent-C programmers to custom design systems quickly. Accessory products include antennae, cables, and packaging.

Specifications of Mica mote are as follows:

- **Processor**: Atmel ATmega 128L
- **Frequency Range**: 902 to 928 MHz - 433.1 to 434.8 MHz
- **Nonvolatile Memory**: 512 KB Since these motes have a small memory, running complex calculations on them is not possible.
- **Attached AA Battery Pack**

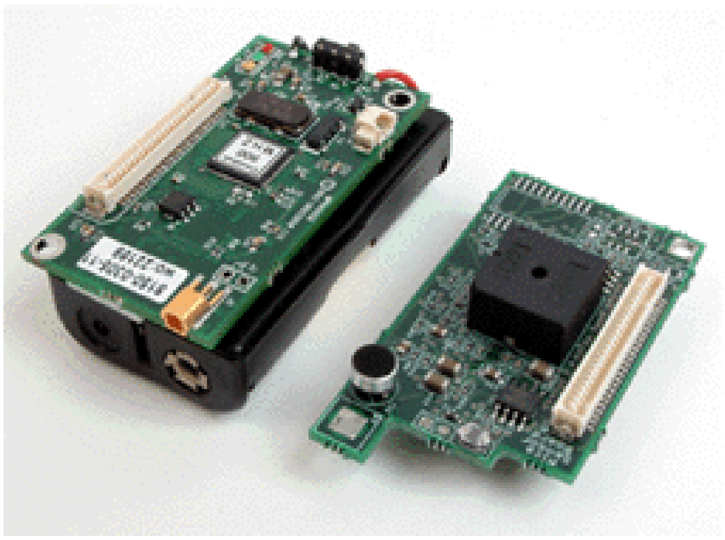


Figure 3.1: Crossbow's mica2 mote and sensor

There is a trade of between the range and power consumption of these motes. The limitation on range of these motes results in fewer neighbors for a mote and slower dissemination of data. These motes require 3rd Generation, Tiny, Wireless Smart Sensors, Plug-in Sensor boards: Light, Temperature, Acceleration/Seismic, Acoustic, and magnetic sensors These sensors cannot sense more than one of the given opportunities Crossbow ships three Mote Processor/Radio module families - MICAz (MPR2400), MICA2 (MPR400), and MICA2DOT (MPR500). The MICAz radio works on the global 2.4GHz ISM band and supports IEEE802.15.4 and ZigBee. The MICA2 and MICA2DOT family is available in 315,433,868/900MHz configurations and support frequency agile operation. These modules are designed for both end-user and OEM applications. All modules provide a processor that runs TinyOS-based code, two-way ISM band radio transceiver, and a logger memory capable of storing up to 100,000 measurements. In addition, these boards offer enhanced processor capabilities, including a boot-loader that allows for over-air reprogramming of Mote code. Wireless Measurement System, MICA2 :

- Has 3rd Generation, Tiny, Wireless Smart Sensors
- TinyOS - Unprecedented Communications and Processing
- 1yr Battery Life on AA Batteries (Using Sleep Modes)
- Wireless Communications with Every Node as Router Capability
- 433, 868/916, or 310 MHz Multi-Channel Radio Transceiver
- Light, Temperature, RH, Barometric Pressure, Acceleration/Seismic, Acoustic, Magnetic, GPS, and other Sensors available

Applications

- Wireless Sensor Networks
- Security, Surveillance, and Force Protection
- Environmental Monitoring

- Large Scale Wireless Networks
- Distributed Computing Platform

3.2 Software Specifications

- **TinyOS and NesC**: TinyOSis an open source operating system designed for wireless sensor networks that supports Crossbow Mica motes, Mica2 motes and Mica2Dot motes and a few other wireless sensor devices. TinyOS is embedded in Crossbow motes and therefore we will use TinyOS structures for developing the system. Programming language used to develop software on TinyOS is nesC. nesC is an extension to C that involves the necessary structures and concepts to support event-driven execution of TinyOS and this project will be implemented using nesC programming language.
- **TinyDB**: TinyDB which is query processing system for extracting information from a network of TinyOS sensors In addition to the mote software, TinyDB provides a PC interface written in Java. JDK 1.3 or later is required. The TinyDB software is to be modified to encoorporate the localization algorithm.
- **Java**: The packet information retrieval, localization algorithm and frontend display along with the topology and the locations is coded in java.

3.2.1 TinyOS

It [14] is an open-source operating system designed for wireless embedded sensor networks. It features a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools - all of which can be used as-is or be further refined for a custom application. TinyOS's event-driven execution model enables fine-grained power management yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces. TinyOS has been ported to over a dozen platforms and numerous sensor boards. A wide community uses it in simulation to develop and test various algorithms and protocols. New releases see over 10,000 downloads. Over 500 research groups and companies are using TinyOS on the Berkeley/Crossbow Motes. Numerous groups are actively contributing code to the sourceforge site and working together to establish standard, interoperable network services built from a base of direct experience and honed through competitive analysis in an open environment.

3.2.1.1 TinyOS Application

Some final comments about a TinyOS application are due and their implications. Since everything in a TinyOS application is static:

- No Dynamic Memory (no malloc)
- No Function Pointers
- No Heap

This means that just about everything is known at compile time by the nesC compiler. This allows the compiler to perform global compile time analysis to detect data race conditions, and where function inlining will improve performance. This relieves the developer of these burdens and hence development is made easier and the systems robustness is improved. The memory map of a TinyOS application is similar to the structure of an executable image on the Unix OS. The Harvard architecture of a COTS Mote (i.e. the Atmel MCU) partitions the memory into two segments: the static program flash memory and the dynamic data SRAM. Each segment has it's own bus. The advantage of this architecture is it allows data and executable code to be fetched in parallel and allows many instructions to executed in one CPU cycle. The Mica2 generation of COTS Motes (our case) consists of 128k of program flash and 4k of SRAM. The memory image is as follows: In the 128K Program Flash

- ”text” section - Executable Code
- ”data” section - Program Constants

In the 4K SRAM

- ”bss” section - Variables
- The rest of the bss is free space - fixed (no dynamic memory)
- stack - grows down in the free space

3.2.2 nesC

nesC [13](pronounced “NES-see”) is an extension to the C programming language designed to embody the structuring concepts and execution model of TinyOS .TinyOS is an event-driven operating system designed for sensor network nodes that have very limited resources (e.g., 8K bytes of program memory, 512 bytes of RAM). The basic concepts behind nesC are:

- *Separation of construction and composition*: programs are built out of components, which are assembled (”wired”) to form whole programs. Components have internal concurrency in the form of tasks. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt.
- *Specification of component behaviour in terms of set of interfaces*: Interfaces may be provided or used by components. The provided interfaces are intended to represent the functionality that the component provides to its user, the used interfaces represent the functionality the component needs to perform its job.
- *Interfaces are bidirectional*: they specify a set of functions to be implemented by the interface's provider (commands) and a set to be implemented by the interface's user (events). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the sendDone event.

- Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts.
- Components are statically linked to each other via their interfaces. This increases runtime efficiency, encourages rubust design, and allows for better static analysis of programs.
- nesC is designed under the expectation that code will be generated by whole-program compilers. This should also allow for better code generation and analysis.

3.2.3 TinyDB

TinyDB [12] is a query processing system for extracting information from a network of TinyOS http://webs.cs.berkeley.edu/tos sensors. Unlike existing solutions for data processing in TinyOS, TinyDB does not require to write embedded C code for sensors. Instead, TinyDB provides a simple, SQL-like interface to specify the data, along with additional parameters, like the rate at which data should be refreshed - much as posing queries against a traditional database. Given a query specifying data interests, TinyDB collects that data from motes in the environment, filters it, aggregates it together, and routes it out to a PC. TinyDB does this via power-efficient in-network processing algorithms. To use TinyDB, its TinyOS components are installed onto each mote in the sensor network. TinyDB provides a simple Java API for writing PC applications that query and extract data from the network; it also comes with a simple graphical query-builder and result display that uses the API. The primary goal of TinyDB is to make work as a programmer significantly easier, and allow data-driven applications to be developed and deployed much more quickly than what is currently possible. TinyDB frees from the burden of writing low-level code for sensor devices, including the (very tricky) sensor network interfaces. Some of the features of TinyDB include:

- **Metadata Management**: TinyDB provides a metadata catalog to describe the kinds of sensor readings that are available in the sensor network.
- **High Level Queries**: TinyDB uses a declarative query language that describe the data wanted, without requiring to say how to get it. This makes it easier to write applications, and helps guarantee that the applications continue to run efficiently as the sensor network changes.
- **Network Topology**: TinyDB manages the underlying radio network by tracking neighbors, maintaining routing tables, and ensuring that every mote in the network can efficiently and (relatively) reliably deliver its data to the user.
- **Multiple Queries**: TinyDB allows multiple queries to be run on the same set of motes at the same time. Queries can have different sample rates and access different sensor types, and TinyDB efficiently shares work between queries when possible.
- **Incremental Deployment via Query Sharing**: To expand the TinyDB sensor network, it is only required to simply download the standard TinyDB code to new motes, and TinyDB does the rest. TinyDB motes share queries with each other; when a mote hears a network message for a query that it is not yet running, it

automatically asks the sender of that data for a copy of the query, and begins running it. No programming or configuration of the new motes is required beyond installing TinyDB.

3.2.3.1 System Overview

This section provides a high level overview of the architecture of the TinyDB software. It is designed to be accessible to users of the TinyDB system who are not interested in the technical details of the system's implementation. We begin with a short description of a typical use-case for TinyDB. Imagine that Mary wishes to locate an unused conference room in her sensor-equipped building, and that an application to perform this task has not already been built. The motes in Mary's building have a sensor board with light sensors and microphones and have been programmed with a room number. Mary decides that her application should declare a room in-use when the average light reading of all the sensors in a room are above . Mary wants her application to refresh this occupancy information every 5 minutes. Without TinyDB, Mary would have to write several hundred lines of custom embedded C code to collect information from all the motes in a room, coordinate the communication of readings across sensors, aggregate these readings together to compute the average light and volume, and then forward that information from within the sensor network to the PC where the application is running. She would then have to download her compiled program to each of the motes in the room. Instead, if the motes in Mary's building are running TinyDB, she can simply pose the following SQL query to identify the rooms that are currently in-use:

```
SELECT roomno, AVG(light), AVG(volume)
FROM sensors
GROUP BY roomno
HAVING AVG(light) >
EPOCH DURATION 5min
```

TinyDB translates this query into an efficient execution plan which delivers the set of occupied rooms every 5 minutes. Mary simply inputs this query into a GUI - she writes no C code and is freed from concerns about how to install her code, how to propagate results across multiple network hops to the root of the network, how to power down sensors during the time when they are not collecting and reporting data, and many other difficulties associated with sensor-network programming. . The system can be broadly classified into two subsystems: Sensor Network Software: This is the heart of TinyDB, although most users of the system should never have to modify this code. It runs on each mote in the network, and consists of several major pieces:

- **Sensor Catalog and Schema Manager**: The catalog is responsible for tracking the set of attributes, or types of readings (e.g. light, sound, voltage) and properties (e.g. network parent, node ID) available on each sensor. In general, this list is not identical for each sensor: networks may consist of heterogeneous collections of devices, and may be able to report different properties.

- **Query Processor**: The main component of TinyDB consists of a small query processor. The query processor uses the catalog the fetch the values of local attributes, receives sensor readings from neighboring nodes over the radio, combines and aggregates these values together, filters out undesired data, and outputs values to parents.

- **Memory Manager**: TinyDB extends TinyOS with a small, handle-based dynamic memory manager
- **Network Topology Manager**: TinyDB manages the connectivity of motes in the network, to efficiently route data and query sub-results through the network.
- **Java-based Client Interface**: A network of TinyDB motes is accessed from a connected PC through the TinyDB client interface, which consists of a set of Java classes and applications. These classes are all stored in the tinyos-1.x/tools/java/tinyos/tinydb package in the source tree.

Major classes include:

- A network interface class that allows applications to inject queries and listen for results
- Classes to build and transmit queries
- A class to receive and parse query results
- A class to extract information about the attributes and capabilities of devices
- A GUI to construct queries
- A graph and table GUI to display individual sensor results
- A GUI to visualize dynamic network topologies
- An application that uses queries as an interface on top of a network of sensors

3.2.3.2 Installation and Requirements

TinyDB requires a basic TinyOS installation, with a working Java installation (and javax.comm library). It is currently designed to work with the nesC compiler (next generation C-like language for TinyOS) and avr-gcc 3.3.The most recent version of TinyDB is always available from the TinyOS SourceForge repository.

3.2.3.3 Running the TinyDBMain GUI

The TinyDBMain Java application provides a graphical interface for distributing queries over motes and collecting data from them. To run this application,

- cd tinyos-1.x/tools/java/net/tinyos/tinydb
- make
- cd tinyos-1.x/tools/java
- java net.tinyos.tinydb.TinyDBMain

Two windows should appear; one, the command window allows to send a variety of control commands to the motes. The other, the query window allows to build and send queries into the network. We will be focusing on the operation of the query window in the next section; the command window is fairly self-explanatory. The query window contains a Display Topology button to show the network topology. This button actually generates a particular query that is executed by the motes, with results displayed in a special visualization. It is a good idea to display network topology and make sure that all motes are alive and communicating.

3.2.3.4 Using TinyDB

TinyDB provides a high-level, declarative language for specifying queries. Declarative languages are advantageous for two reasons. First, they are relatively easy to learn, with queries that are easy to read and understand. Second, they allow the underlying system to change how it runs a query, without requiring the query itself to be changed. This is important in a volatile context like sensor networks, where the best underlying implementation may need to change frequently - e.g. when motes move, join or leave the network, or experience shifting radio interference. In TinyDB, the execution strategy for a user query can change each time the query is run, or even while the query runs, without any need for re-typing the query or recompiling an application that embeds the query. Before describing TinyDB's query facilities, a few words on TinyDB's data model are in order. TinyDB implicitly queries one single, infinitely-long logical table called sensors. This table has one column for each attribute in the catalog, including sensor attributes, nodeIDs, and some additional "introspective" attributes (properties) that describe a mote's state. This table conceptually contains one row for each reading generated by any mote, and hence the table can be thought of streaming infinitely over time. A given mote may not be able to generate all the attributes, e.g., if it does not have the sensor that generates the attribute. In that case, the mote will always generate a NULL value for that attribute. TinyDB's query language is based on SQL, and it will be referred as TinySQL. As in SQL, queries in TinySQL consist of a set of attributes to select (e.g. light, temperature), a set of aggregation expressions for forming aggregate result columns, a set of selection predicates for filtering rows, and optionally a grouping expression for partitioning the data before aggregation. Aggregation is commonly used in the sensor environment. Currently, TinySQL results are very similar to SQL, in that they are based on snapshots in time - that is, they are posed over rows generated by multiple sensors at one point in time. Temporal queries that combine readings over several time periods are not supported in the current release. Instead, TinySQL runs each query repeatedly, once per time-period or "epoch". The duration of an epoch can be specified as part of a TinySQL query; the longer the duration, the less frequent the results, and the less drain on the mote batteries.

When using TinyDB, it is also possible to write queries by hand, either by using the "Text Interface" pane of the the GUI (which can be brought up by default by using the command-line argument "-text"), or via the SensorQueryer.translateQuery API call. We assume here that the reader has a familiarity with the basics of SQL. A number of books and websites provide simple SQL tutorials. No deep knowledge of SQL is required to use TinyDB; the basics will do. The simplest way to learn TinySQL is to use the graphical query builder. However, we also provide a simple, informal description of the syntax here. TinyDB provides an SQL-like query language, which is simplified in a number of ways, but which also provides some new sensor-specific syntax. TinySQL queries all have the form:

```
SELECT select-list
[FROM sensors]
WHERE where-clause
[GROUP BY gb-list
[HAVING having-list]]
[TRIGGER ACTION command-name[(param)]]
[EPOCH DURATION integer]
```

The SELECT, WHERE, GROUP BY and HAVING clauses are very similar to the functionality of SQL. Arithmetic expressions are supported in each of these clauses. As in standard SQL, the GROUP BY clause is optional, and if GROUP BY is included the HAVING clause may also be used optionally.

3.2.3.5 The TinyDB Java API

The API contains a number of objects encapsulating the TinyDB network, the TinyDB catalog, the construction of TinyDB queries, and the manner in which the application listens for and interprets query results. These objects appear in the corresponding .java files in tinyos-1.x/tools/java/tinyos/tinydb.

1. **TinyDBNetwork**: This object is the main interface to a network of motes. It is responsible for injecting new queries into the network (sendQuery()), for cancelling queries (abortQuery()), and for providing results from the network to multiple query “listeners”. Only one instance of the TinyDBNetwork object needs to be allocated for a network; that instance can manage multiple ongoing queries, and multiple listeners. Each query’s output can be sent to multiple listeners, and each listener can listen either to a single query, or to all queries. Internally, the object maintains a list of live queries, and three sets of listeners: processedListeners are signed up for a specific query ID, and get a stream of final (“processed”) answer tuples for that query. qidListeners are signed up for a specific query ID, and get copies of all messages that arrive for that query. These messages may not be final query answers. They may be individual attributes from an answer tuple, or unaggregated sub-result tuples. listeners are signed up to receive a copy of all unprocessed messages for all queries. The various listeners can be added or deleted to the object on the fly via addResultListener() and removeResultListener() - note that different arguments to the addResultListener method result in one of the 3 different kinds of listeners above. The TinyDBNetwork object handles all incoming AMAM messages from the serial port, and dispatches copies of them to the listeners and qidListeners accordingly. It also processes the messages to generate result tuples (via QueryResult.MergeQueryResult()) and sends them to processedListeners accordingly. As part of processing results, it maintains info on epochs to make sure that the epoch semantics of the results are correct. Internally, the TinyDBNetwork object also has a background thread that participates in the sensor network’s routing algorithms. It periodically sends information down the routing tree, so that children know to choose the root as a parent, and so that children can decide how to share the timeslots in an epoch.

2. **SensorQueryer**: This class appears in the parser subdirectory. It represents a simple parser for TinySQL. The main method of interest is translateQuery, which takes an SQL string and returns a corresponding TinyDBQuery object, which we proceed to describe next.

3. **TinyDBQuery**: This is a Java data structure representing a query running (or to be run) on a set of motes. Queries consist of:

   – a list of attributes to select

   – a list of expressions over those attributes, where an expression is

     ∗ a filter that discards values that do not match a boolean expression

     ∗ an aggregate that combines local values with values from neighbors, and optionally includes a GROUP BY column.

  – an SQL string that should correspond to the expressions listed above.

  In addition to allowing a query to be built, this class includes handy methods to generate specific radio messages for the query, which TinyDBNetwork can use to distribute the query over the network, or to abort the query. It also includes a support routine for printing the query result schema.

4. **QueryResult**: This object accepts a query result in the form of an array of bytes read off the network, parses the results based on a query specification, and provides a number of utility routines to read the values back. It also provides the mergeQueryResult functionality for processedListeners. This does concatenation of multiple aggs as separate attributes of a single result tuple, and finalizes aggregates, by combining data from multiple sensors.
5. **AggOp**: This provides the code for the aggregation operators SUM, MIN, MAX, and AVG. It includes representation issues (internal network codes for the various ops, and code for pretty-printing), and also the logic for performing final merges for each aggregate as part of QueryResult:MergeQueryResult().
6. **SelOp**: This provides the logic for selection predicates. Currently this includes representations for simple arithmetic comparisons (internal network codes for the arithmetic comparators, and pretty-printing.)
7. **Catalog**: This object provides a very simple parser for a catalog file - it reads in the file, and after parsing it provides a list of attributes.
8. **CommandMsgs**: This is a class with static functions to generate message arrays that can be used to invoke commands on TinyDB motes.

3.2.3.6 The TinyDB Demo Application

The TinyDB application allows users to interactively specify queries and see results. It also serves as an example of an application that uses the TinyDB API. As with traditional database systems, it is expected that many programmers will want to embed queries within more specialized application code. Such programmers can look at the TinyDB application for an example of how this is done. The TinyDB application consists of only a few objects:

1. **TinyDBMain**: This is the main loop for the application. It opens an Active Message (AM) connection to the Serial Port ("COM1"), and uses it to initialize a TinyDBNetwork object. It allocates the GUI objects CmdFrame and QueryFrame for the application, which issue queries and in turn generate visualizations of results. There are also some simple wrapper routines for the TinyDBNetwork methods to add and remove queries from listeners.
2. **CmdFrame** : This is a simple GUI for sending TinyDB commands (from the CommandMsgs API object) into the network.
3. **MainFrame** : This is the main GUI for building queries with TinyDB. It provides a simple API for generating new query ID's and processing keyboard input. The buttons along the right send either send the current query being built ("Send Query") into the network for execution, or execute a predefined query, as follow:

  - **Display Topology**: A visualization of the network topology, which is extracted from the network via a standard TinyDB query.
  - **Mag. Demo**: A visualization of magnetometers laid out in a fixed grid. This is an example of simple demo application that can run on TinyDB: in this case, TinyDB is used to identify sensors with magnetometer readings greather than some threshold to detect metallic objects moving through a grid of motes. The major portion of the GUI contains a tabbed pane that provides two different interfaces for inputting queries:
  - **GuiPanel**: A graphical query builder to construct a valid TinyDBQuery object and send it into the network via TinyDBNetwork.sendQuery() , In addition to allowing users to specify ad hoc queries, it provides a button to send off two pre-prepared queries that have special visualizations:
  - **TextPanel**:A textual query editor that allows queries to be input in TinySQL language.

4. **QueryField**: Simple support routines for handling attributes in the query builder.
5. **ResultFrame**: ResultFrame displays a scrolling list with results from queries in it, side-by-side with a graph of query results when such results are available. For each query, it adds a processedListener to the TinyDBNetwork in order to receive the results, which it plots via ResultGraph.
6. **ResultGraph**: A simple wrapper for the plot package, to interactively graph query results.
7. **Plot**: A graph-plotting package from the Ptolemy project.
8. **Topology**: A set of classes for constructing the TinyDB network-topology-extraction query, and for displaying the results as a (dynamic) topology graph.

3.2.3.7 TinyDB Source Files

The following files in the TinyOS CVS tree are a part of the TinyDB distribution:
tinyos-1.x/tos/lib/TinyDB

- /AggOperator.rd
- /DBBufferC.nc
- /DBBuffer.nc
- /DBBuffer.h
- /ExprEvalC.nc
- /ExprEval.nc
- /NetworkC.nc
- /Network.nc
- /Operator.nc
- /ParsedQueryIntf.nc

- /ParsedQuery.nc
- /QueryIntf.nc
- /Query.nc
- /RadioQueue.nc
- /SelOperator.nc
- /TinyDBAttr.nc
- /TinyDBCommand.nc
- /TinyDB.h
- /TupleIntf.nc
- /TupleRouter.nc
- /TupleRouterM.nc
- /Tuple.nc

tinyos-1.x/tos/interfaces

- /Attr.h
- /AttrRegisterConst.nc
- /AttrRegister.nc
- /AttrUse.nc
- /Command.h
- /CommandRegister.nc
- /CommnadUse.nc
- /MemAlloc.nc
- /SchemaType.h

tinyos-1.x/tos/lib

- /Command.nc
- /Attr.nc
- /TinyAlloc.nc

tinyos-1.x/tools/java/net/tinyos/tinydb

- AggExpr.java
- AggOp.java
- Catalog.java

- CmdFrame.java
- CommandMsgs.java
- MagnetFrame.java
- QueryExpr.java
- QueryField.java
- QueryListener.java
- QueryResult.java
- ResultFrame.java
- ResultGraph.java
- ResultListener.java
- SelExpr.java
- SelOp.java
- TinyDBCmd.java
- TinyDBMain.java
- TinyDBNetwork.java
- TinyDBQuery.java

Makefile

- parser/
  - Makefile
  - senseParser.cup,lex
- tinyos-1.x/apps/TinyDBApp
  - Makefile
  - TinyDBApp.nc

3.3 Functional Requirements

The system should be able to

- Fix the positions of the desired beacon motes
- Obtain the neighbours of each sensor in the network
- Based on the knowledge of the positions of beacons and identification of neighbours, calculate all possible positions of each sensor in the network i.e localize their positions to a fixed set.
- In case of a dynamic system where positions vary constantly, the new locations of each sensor should also be calculated within a fixed period of time.

3.3.1 Non Functional Requirements

3.3.2 Scalability

Wireless sensor networks generally contain hundreds or even thousands of individual nodes. It is a challenge to show the required performance in functioning of the system under such dense networks.

3.3.3 Efficient Data Propagation

Wireless sensor network applications are generally monitoring applications. Most of these applications require delivery of data in real-time. Therefore, our system should be efficient while ensuring that it is without extended delays and overheads.

3.3.4 Memory Efficiency

Memory available to the sensors is in the order of KBs, some of which is already filled up by OS and other applications. Therefore, applications and application middlewares should be very carefully designed to avoid extensive use of available memory

Chapter 4

Design

4.1 Data Flow Diagram

The Data Flow Diagram gives an overall view of the process:

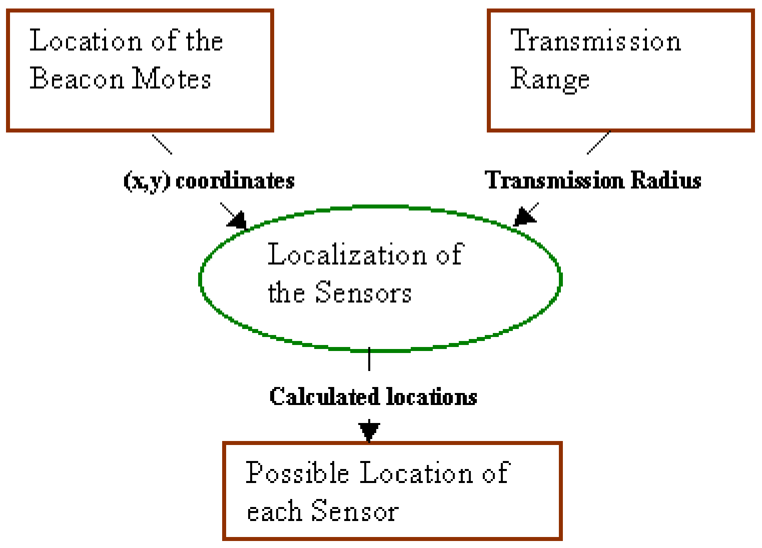

Location of the Beacon Motes
Transmission Range
(x,y) coordinates
Transmission Radius
Localization of the Sensors
Calculated locations
Possible Location of each Sensor

Figure 4.1: 0-Level DFD

4.2 Basic Design

In many applications of wireless sensor networks sensors are deployed un-tethered in hostile environments. For location aware in these applications, it is essential to ensure that sensors can determine their location, even in the presence of malicious adversaries. Many sensor network applications require location awareness, but it is often too expensive to include a GPS receiver in a sensor network node. Hence, localization schemes for sensor networks typically use a small number of seed nodes that know their location and protocols whereby other nodes estimate their location from the messages they

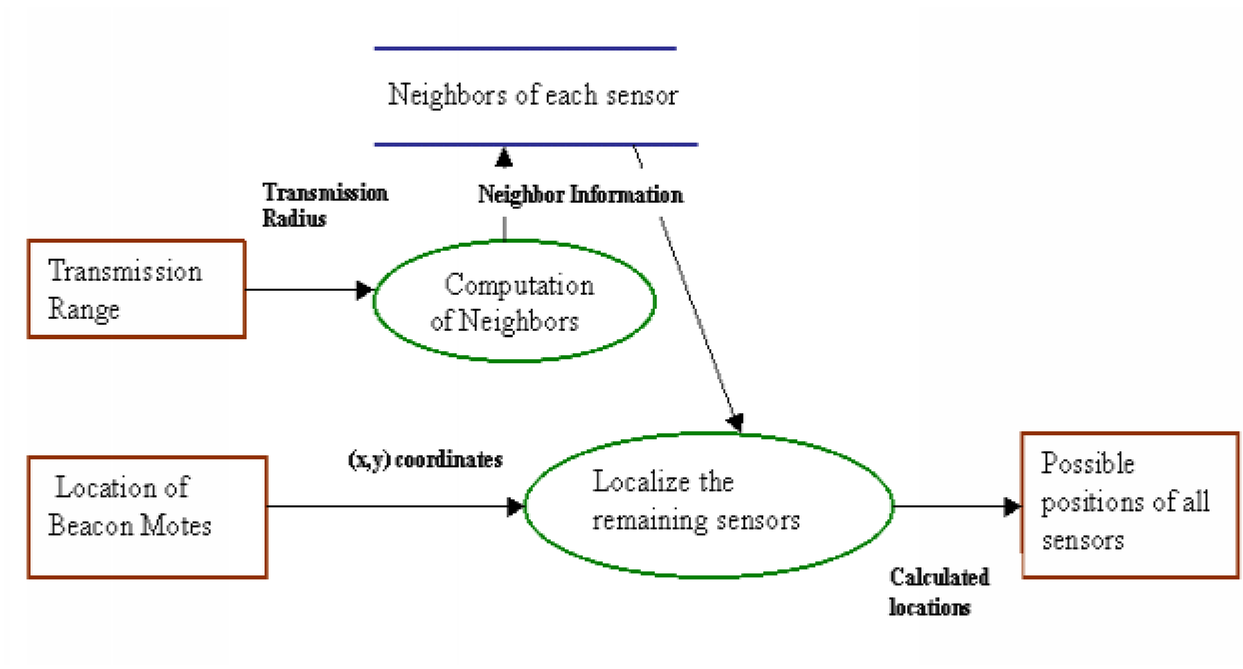


Figure 4.2: 1-Level DFD

receive.In this section we address the problem of enabling sensors of wireless sensor networks to determine their location in a 2-dimensional environment.

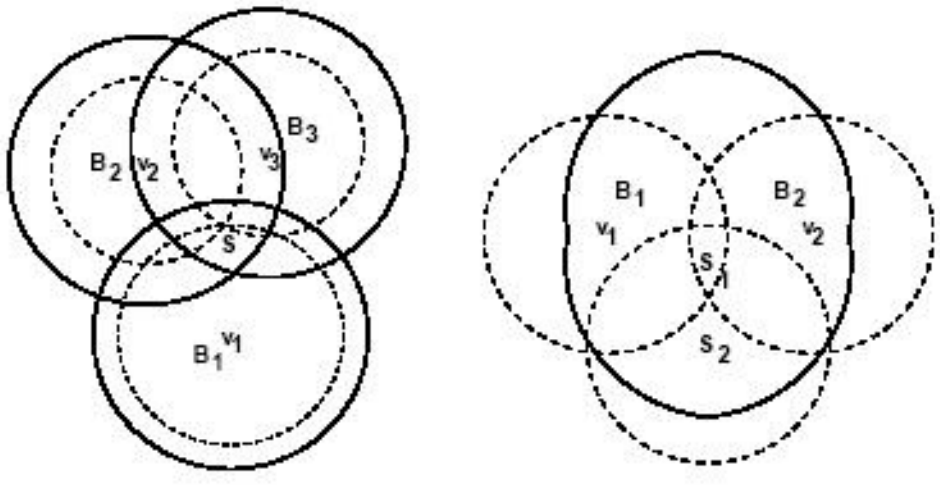


Figure 4.3: Localization with IR

Let R denote the range, and D(x,r) a disk of radius r centred at x. The figure shows S_1 and S_2 which are two sensors with unknown locations. S_1 is in the range of beacons B_1 and B_2; Bi located at v_i. Therefore it gets localized to the region of intersection of $D(v_i, R), i = 1, 2$ shown by dotted circles which centres v_1 and v_2. S_2, though not in the range of either B_1 or B_2, is in the range of S_1; the dotted circle centred at S_1 is $D(v_{S1}, R)$ where v_{S1} denotes the location of S_1. Therefore S_2 gets localized to the region bounded by the solid curve in figure. In this way, sensors can learn and improve localization sets iteratively as discussed in [1].

4.3 Algorithm

The algorithm to localize sensors in an ad-hoc network using the in-range method:

1. The positions of beacon motes (the motes whose positions are known) are initialized by setting the locX and locY attributes. The entire test bed area is taken as the initial set of positions for the other motes, while the beacon motes have only one point in their location set.

2. The neighbors of each sensor are computed as follows:

   – Each sensor broadcasts a signal at regular intervals

   – The immediate neighbors respond back. The neighbor buffer of the sender sensor is updated each time it receives a response.

   – The neighbor information is routed to the main PC via the multi-hop network

   – The neighbor buffer of each sensor is refreshed periodically (to facilitate dynamic changes)

3. For each sensor, possible locations represented by a set are reckoned by the intersection of points in the transmission range of each of its neighbor.

4. If a node is not a neighbor of a sensor, the set of points representing the range of the non-neighbor is subtracted from possible locations obtained in step 3. This is done because, if a node is a non-neighbor of a sensor, the latter cannot lie in the range of the former.

5. The intersection of the previous set (calculated in the previous iteration) with that of possible locations computed from the locations of the neighbors and non-neighbors (from step 4) give the final set of possible locations of the sensor. The number of possible locations of each sensor is non- increasing over iterations.

6. Steps 3 , 4 and 5 are repeated, a chosen number of times till the possible positions of a sensor is narrowed down to a set of n possibilities, which does not change over further iterations, or by fixing the number of iterations to a number “n”.

Chapter 5

Testing and Results

When working with embedded devices, it is very difficult to debug applications. Because of this, we have to make sure that the tools that we are using are working properly and that the hardware is functioning correctly. This will save countless hours of searching for bugs in the application when the real problem is in the tools.

5.1 White Box Testing

5.1.1 Unit Testing

Hardware testing involves checking working of the system and the hardware.

5.1.1.1 TinyOS Installation Verification

A TinyOS development environment requires the use of avr gcc compiler, perl, flex, cygwin if you use windows operation system, and JDK 1.3.x or above. TinyOS provides a tool named toscheck to check if the tools have been installed correctly and that the environment variables are set.
First, we run toscheck (it should be in the current path - a copy is also in tinyos-1.x/tools/scripts). The expected output is as follows:

```
toscheck
Path:
/usr/local/bin
/usr/bin
/bin
/cygdrive/c/jdk1.3.1_01/bin
/cygdrive/c/WINDOWS/system32
/cygdrive/c/WINDOWS
/cygdrive/c/avrgcc/bin
.

Classpath:
/c/alpha/tools/java:.:/c/jdk1.3.1_01/lib/comm.jar

avrgcc:
/cygdrive/c/avrgcc/bin/avr-gcc
Version: 3.0.2
```

```
perl:
/usr/bin/perl
Version: v5.6.1 built for cygwin-multi

flex:
/usr/bin/flex

bison:
/usr/bin/bison

java:
/cygdrive/c/jdk1.3.1_01/bin/java
java version "1.3.1_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_01)
Java HotSpot(TM) Client VM (build 1.3.1_01, mixed mode)

Cygwin:
cygwin1.dll major: 1003
cygwin1.dll minor: 3
cygwin1.dll malloc env: 28

uisp:
/usr/local/bin/uisp
uisp version 20010909


toscheck completed without error.
```

5.1.1.2 Hardware verification

To test the hardware, we use an application: MicaHWVerify. It is designed for the purpose of verifying mica/mica2/mica2dot mote hardware only.
In the apps/MicaHWVerify directory , type

```
(mica platform) make mica
(mica2/mica2dot) PFLAGS=-DCC1K_DEF_FREQ=<freq>
  make [mica2|mica2dot]
```

The compilation process must complete without any errors (Compilation for the mica2dot will generate a warning about the SerialID component). If it is compiled correctly, it will print out a profile of the memory used by the application. While the exact build directory and memory footprints will vary depending on the platform, it should look like:

```
compiled MicaHWVerify to build/mica2/main.exe
       10386 bytes in ROM
       390 bytes in RAM
avr-objcopy --output-target=srec build/mica2/main.exe build/mica2/main.srec
```

Next step is to install the application onto a mote. A powered-on node is placed into a programming board. The red LED on the programming board should light. The programming board is connected to the parallel port of the computer. To load the program on to the device, using a parallel port programmer, we type :

```
make reinstall [mica|mica2|mica2dot]
```

The output :

```
installing mica2 binary
uisp -dprog=<yourprogrammer> -dhost=c62b270 -dpart=ATmega128
 --wr_fuse_e=ff --erase --upload if=build/mica2/main.srec
Atmel AVR ATmega128 is found.
Uploading: flash
Fuse Extended Byte set to 0xff
```

This output shows that the programming tools and the computer's parallel port are working.
The next step is to verify the mote hardware. First, confirm that the LEDs are blinking like a binary counter. Next, the programming board is connected to the serial port of the computer. The hardware verify program will send data over the UART that contains it status. To read from the serial port, TinyOS provides a java tool called hardware_check.java. It is located in the same directory. This tool must be built and run. The commands are shown below assuming COM1 at 57.6 KBaud is used to connect to the programming board.

```
make -f jmakefile
MOTECOM=serial@COM1:57600 java hardware_check
```

The output on the PC is:

```
hardware_check started
Hardware verification successful.
Node Serial ID: 1 60 48 fb 6 0 0 1d
```

This program checks the serial ID of the mote (except on the mica2dot), the flash connectivity, the UART functionality and the external clock. If all status checks are positive, the hardware verification successful message is printed on the PC screen.
If any failure report on the monitor is seen, another mote might be needed.

5.1.1.3 Radio Verification

To verify radio, two nodes are needed . The second node (that has passed the hardware check up to this point) is taken and installed with TOSBase. This node acts as a radio gateway to the first node. Once installed, this node is left in the programming board and the original node is placed next to it. The hardware_check java application is re-run. The output should be the same as shown in the previous section (but will display the serial ID of the remote mote). The indication of a working radio system is, again, something like:

```
hardware_check started
Hardware verification successful.
Node Serial ID: 1 60 48 fb 6 0 0 1d
```

If the remote mote is turned off or not functioning, it will return a message ”Node transmission failure”.
If the system and hardware pass all the above tests, TinyOS is ready to be used for building applications.

5.1.1.4 TinyDB Verification

Three motes are required to test the tinydb package. All three are programmed with the TinyDBApp application, setting their id’s to 0, 1, and 2. The motes are turned on and are the mote programmed with id 0 is connected to the PC serial port. (To program a mote with a specific id, run make mica install.nodeid, where nodeid is the id with which the mote is associated.)
The TinyDBMain class in tools/java/net/tinyos/tinydb is used to interact with the motes The java classes are first built; to do this, we need to ensure that several packages are in the CLASSPATH. The packages needed are JLex.jar, cup.jar, and plot.jar; all three are available in tools/java/jars.There is a small program to set your classpath , called ”javapath” in the tools/java/ directory. To use it, value of the CLASSPATH is set to the output of this command (it will prepend the new directories and jars to your current CLASSPATH.) To use it under bash (in Cygwin or Linux), we type:

```
export CLASSPATH=‘path/to/tinyos/tools/java/javapath‘
```

Under sh or csh ”setenv CLASSPATH ...” is written instead of ”export CLASSPATH=...”. Now, the java classes are built by typing the following:

```
cd path/to/tinyos/tools/java/net/tinyos/tinydb
make
```

This may take several minutes and will output lots of text as the TinyDB query parser is compiled. Now, the TinyDB GUI is tested by running it from the tools/java directory;

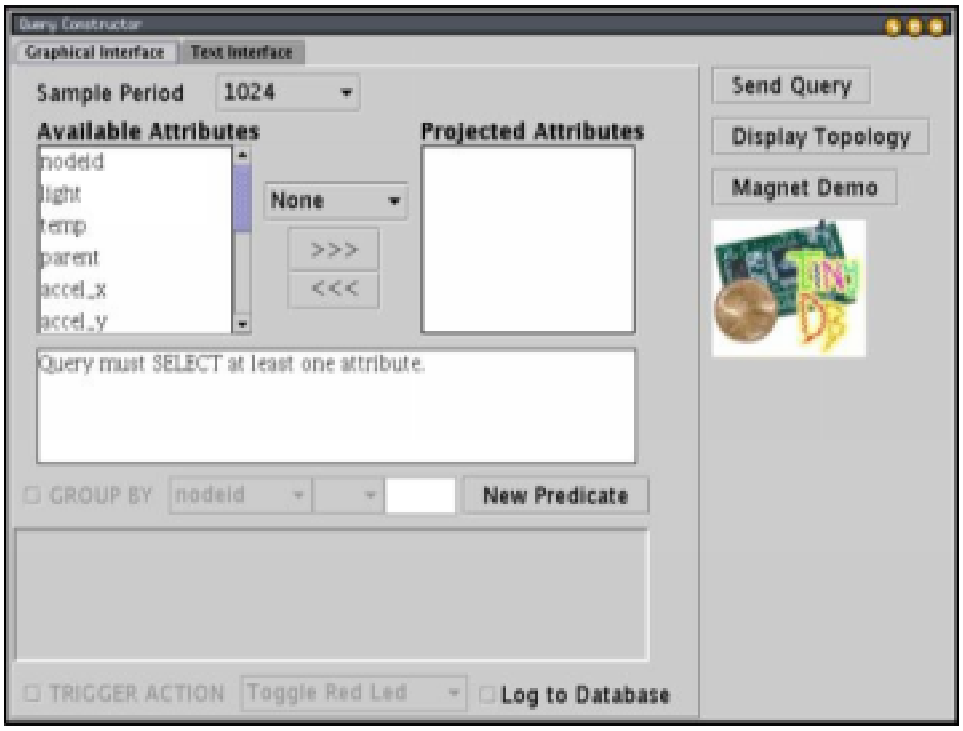
```
cd ../../..
java net.tinyos.tinydb.TinyDBMain
```

The TinyDB GUI should appear.

The test is complete.

5.1.1.5 GetNeighbours module

The module was loaded in each of the sensors, and the code was tested to ensure that every node collects its immediate neigbors accurately. Secondly, the time required to receive the neighbor information from all the motes was noted. Thirdly, to make sure that the module functions accurately in situations, where nodes change their position over time, the code was enhanced to encorporate periodic refreshing of the neighbor buffer table. The module was then tested in a dynamic environment.



Query must SELECT at least one attribute.

Figure 5.1: TinyDB GUI

5.1.1.6 Localization module

The localization algorithm was initially simulated before integrating it with the hardware. The positions of beacon nodes were fixed and the information about the neighbours of each sensor node was fed to the program. The following cases were tested:

1. Localization of a node when it is in the vicinity of beacon nodes.
2. Localization of a node when it is in the vicinity of beacon nodes and other localized nodes.
3. Localization of a node when it is in the vicinity of only other localized nodes.

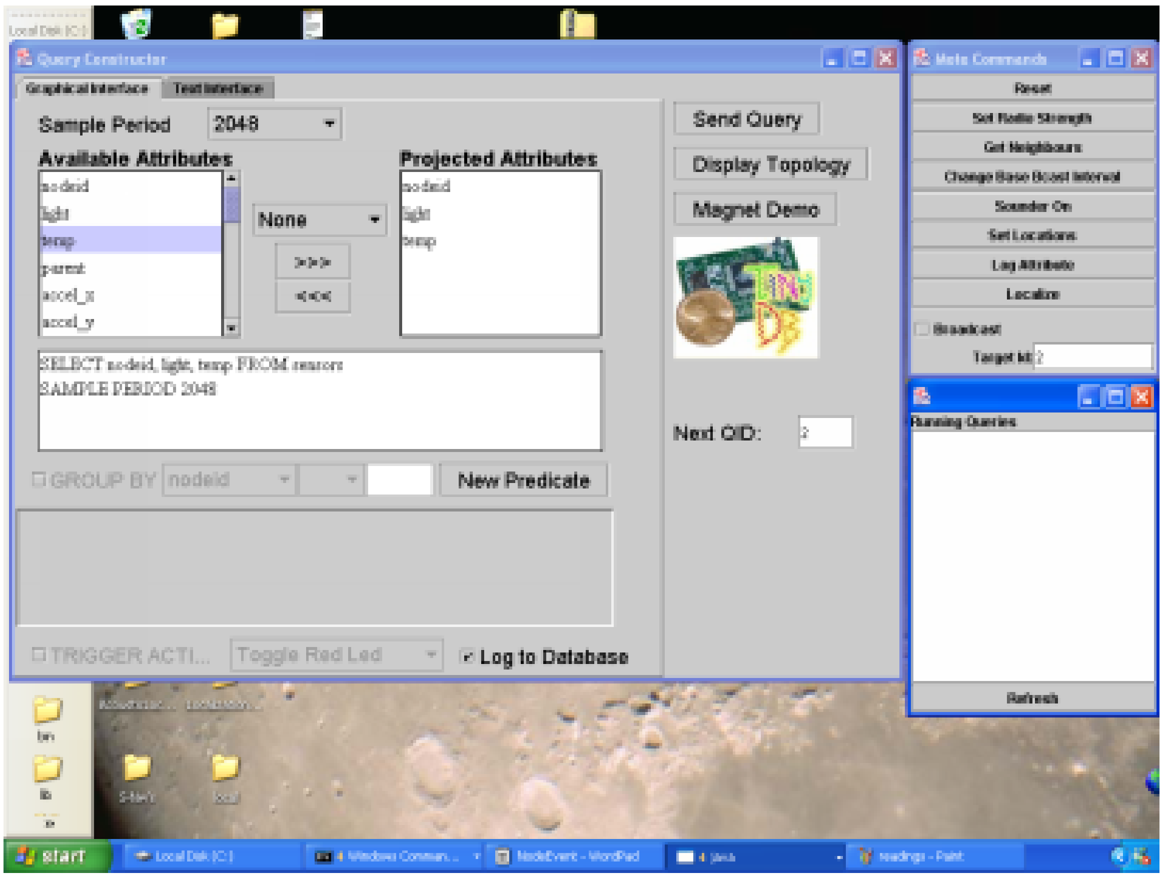
5.1.2 Integrated Testing

The localization algorithm was integrated with the TinyDB application.The integrated application could perform localization and display the localized positions of the sensors in a two-dimensional environment along with the network topology and sensor readings such as temperature, light, sound etc.

The ”Mote commands” panel of the modified frontend has a few new buttons such as Get Neighbours, Set Locations and Localize. The frontend also has new attributes such as locX, locY and nbrs

New functions such as setlocation (for beacons), getneighbors and localization had to be added to the existing TinyDBpackage, integrated and tested.

The TinyDB package is responsible for querying sensor readings from the different motes. It periodically broadcasts queries to the motes, and the motes in turn respond



Sample Period 2048

Available Attributes: nodeid, light, temp, parent, accel_x, accel_y

Projected Attributes: nodeid, light, temp

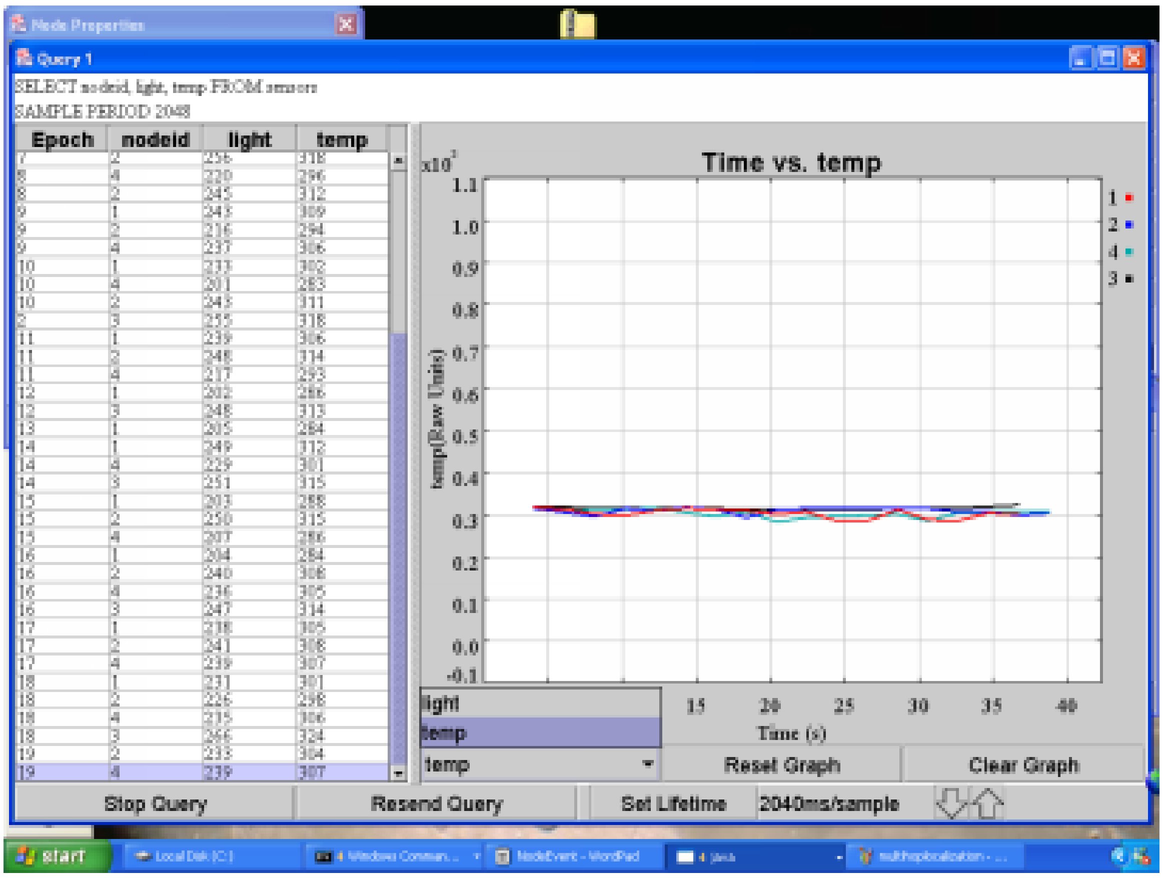
```
SELECT nodeid, light, temp FROM sensors
SAMPLE PERIOD 2048
```

Figure 5.2: Modified frontend to encorporate localization.

to the query requests. When two or more motes respond at the same time, it leads to contention. On integrating the localization algorithm, the contention further increases because the nodes constantly query for their neighbours. The interval between successive neighbour queries was difficult to determine. Repeated testing with different time intervals was performed to determine the ideal interval where contention is minimal and efficiency is maximized.

The same problem was faced when the decision had to be made regarding how often the localization algorithm has to be run. The localization algorithm does not function until it gets the neighbour information from all the nodes. Secondly, due to dynamic changes in the positions of the nodes i.e the positions of the motes might change over time, the localization algorithm should immediately detect a change in location and display the new localized positions of the motes.

In pratical applications, the transmission range is usually around 5 metres. It is not possible to test all cases taking indoors, due to limited space constraints. Hence, the transmission range had to be scaled down to an appropriate value, so that we could cover different possible arrangements and deployment of sensors in the network.



SELECT nodeid, light, temp FROM sensors
SAMPLE PERIOD 2048

Epoch	nodeid	light	temp
7	2	256	318
8	4	220	296
8	2	245	312
9	1	243	309
9	2	216	294
9	4	237	306
10	1	233	302
10	4	201	283
10	2	243	311
2	3	255	318
11	1	239	306
11	2	248	314
11	4	217	293
12	1	202	286
12	3	248	313
13	1	205	284
14	1	249	312
14	4	229	301
14	3	251	315
15	1	203	288
15	2	250	315
15	4	207	286
16	1	204	284
16	2	240	308
16	4	236	305
16	3	247	314
17	1	238	305
17	2	241	308
17	4	239	307
18	1	231	301
18	2	226	298
18	4	235	306
18	3	266	324
19	2	233	304
19	4	239	307

Figure 5.3: Sensor Readings

5.1.3 Results : Black Box Testing

In all the test cases, the transmission range is set to 5 units.

5.1.3.1 Example 1: A mote in the neighborhood of one beacon

Beacon 1: Beacon at (5,5)
Node 2: Node at (7,7) which has to be localized

Node 2 is in the neighborhood of Beacon 1 therefore; Node 2 should be localized to a circle of radius 5 (transmission range) around (5,5).

The output is as shown in the figure 5.6. The red circle corresponds to beacon 1 and '2' in blue indicates the possible positions of node 2. It is noted that location (7,7) is also included in the set. The green lines signify the network topology.

5.1.3.2 Example 2: A mote in the neighborhood of two beacons

Beacon 1: Beacon at (10,5)
Beacon 2: Beacon at (7,7)
Node 3: Node at (7,4) which has to be localized

Node 3 is in the neighborhood of Beacon 1 and Beacon 2. Therefore, Node 3 should be localized to a region in the intersection of circles of radius 5 (transmission range)

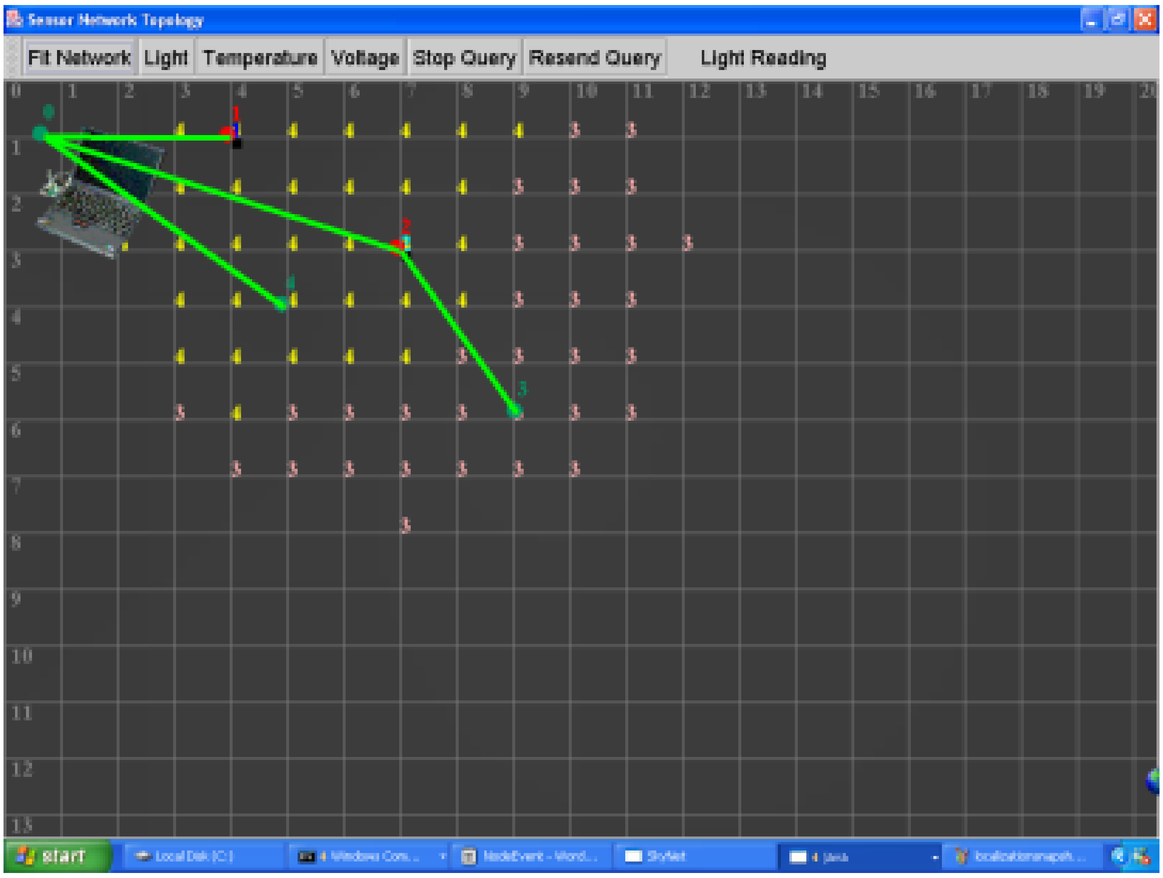


Figure 5.4: Localization: A multihop scenario

around (10,5) and (7,7).
The output is as shown in the figure 5.8. The red circles corresponds to beacon 1 and beacon 2. '3' in pink indicates the possible positions of node 3. It is noted that location (7,4) is also included in the set. The green lines signify the network topology.

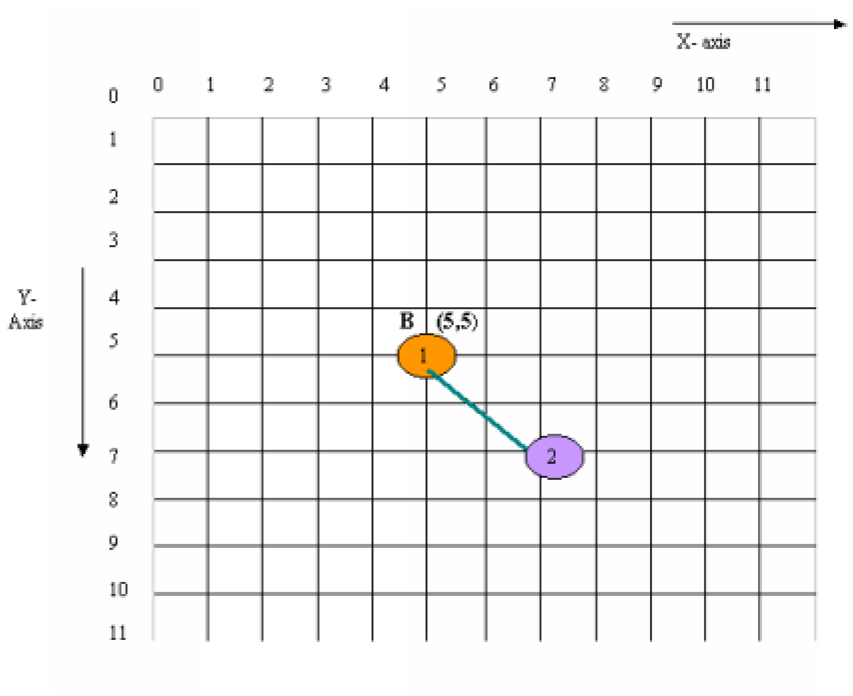


Figure 5.5: Example 1: The actual testbed

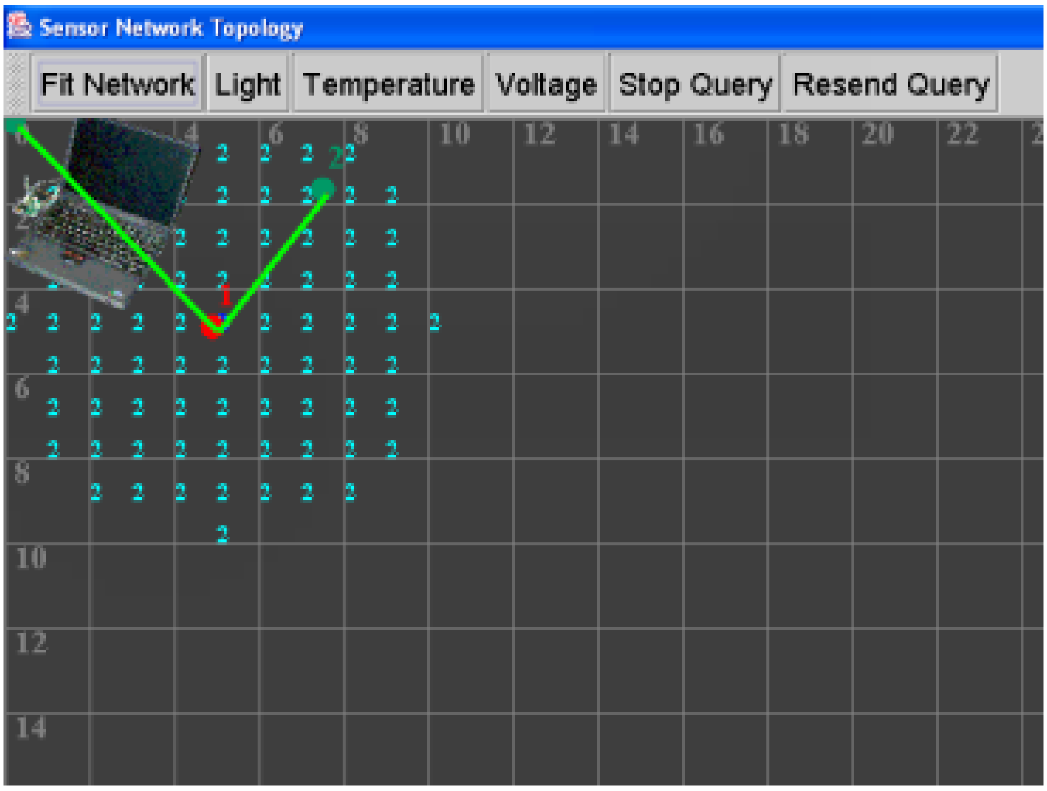


Figure 5.6: Example 1: Output

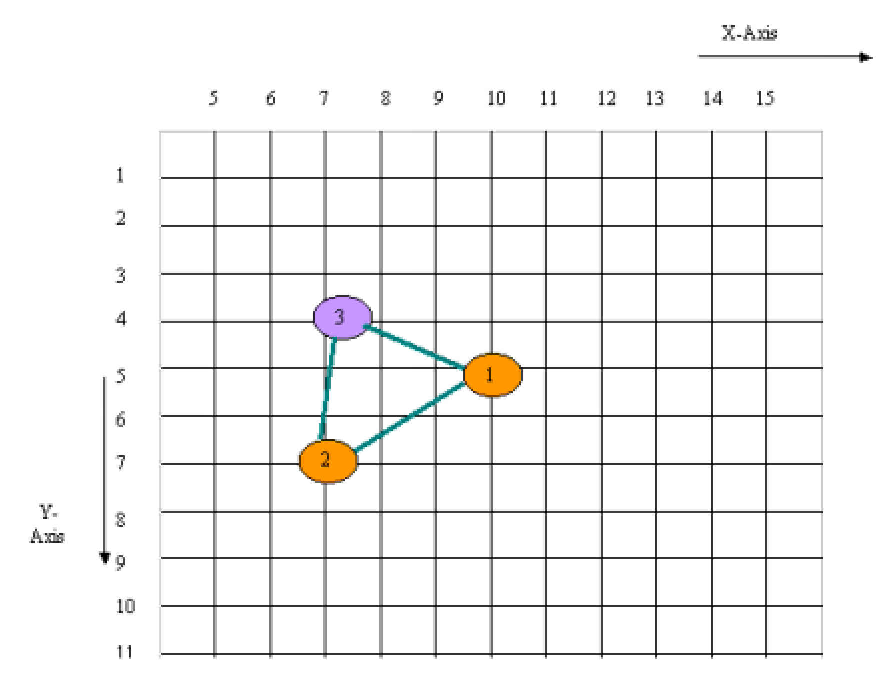


Figure 5.7: Example 2: The actual testbed

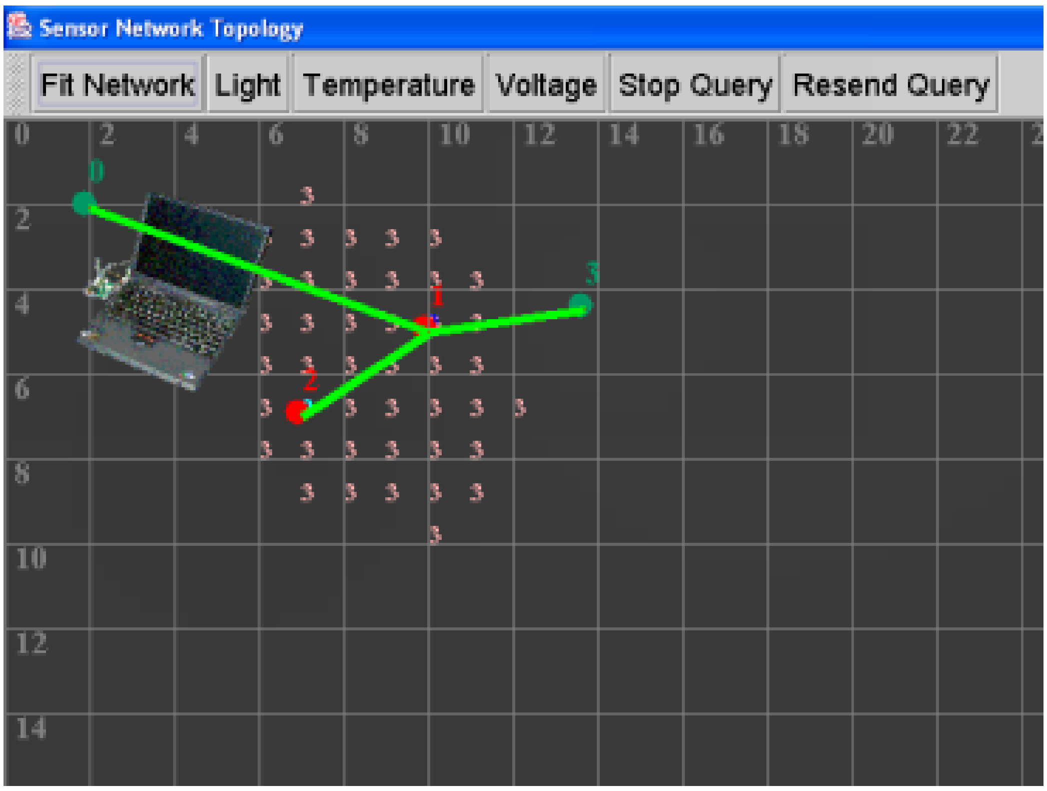


Figure 5.8: Example 2: Output

5.1.3.3 Example 3: A mote in the neighbourhood of one beacon but not the other

Beacon 1: Beacon at (10,5)
Beacon 2: Beacon at (7,7)
Node 3: Node at (13,5) which has to be localized

Node 3 is in the neighborhood of Beacon 1 but not in the neighborhood of Beacon 2. Therefore, Node 3 should be localized to a circle of radius 5 (transmission range) around (10,5) eliminating the points in the circle around (7,7).

The output is as shown in the figure 5.10. The red circles corresponds to beacon 1 and beacon 2. '3' in pink indicates the possible positions of node 3. It is noted that location (13,5) is also included in the set. The green lines signify the network topology.

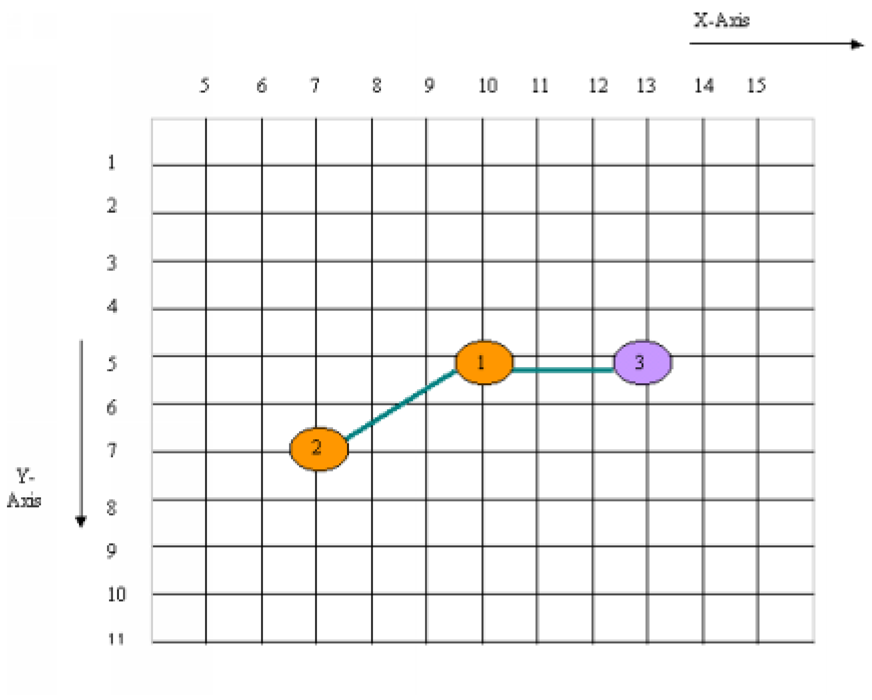


Figure 5.9: Example 3: The actual testbed

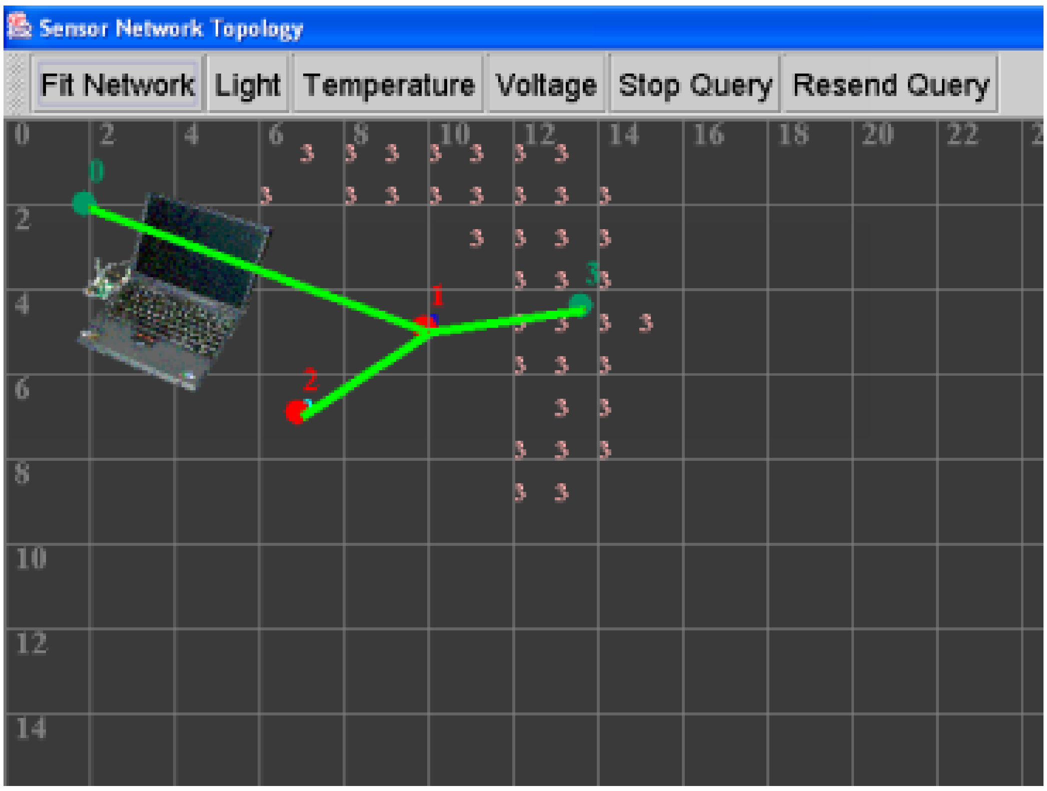


Figure 5.10: Example 3: Output

5.1.3.4 Example 4: A mote in the neighbourhood of no beacons

Beacon 1: Beacon at (10,5)
Beacon 2: Beacon at (7,7)
Node 3: Node at (10,9) which has to be localized
Node 4: Node at (12,11) which has to be localized

Node 3 is in the neighborhood of Beacon 1 and in the neighborhood of Beacon 2. Therefore, Node 3 should be localized to the intersections of circles of radius 5 (transmission range) with (10,5) and (7,7) as centres
Node 4 is only in the neighborhood of Node 3. Therefore Node 4 should be localized to a set of positions in the neighborhood of the localized positions of Node 3, but after eliminating the set of points within a circle of radius 5 around (10,5) and (7,7).

The output is as shown in the figure 5.12. The red circles corresponds to beacon 1 and beacon 2. '3' in pink indicates the possible positions of node 3. It is noted that location (10,9) is also included in the set. '4' in yellow indicates the possible positions of node 4. It is noted that location (12,11) is also included in the set. The green lines signify the network topology.

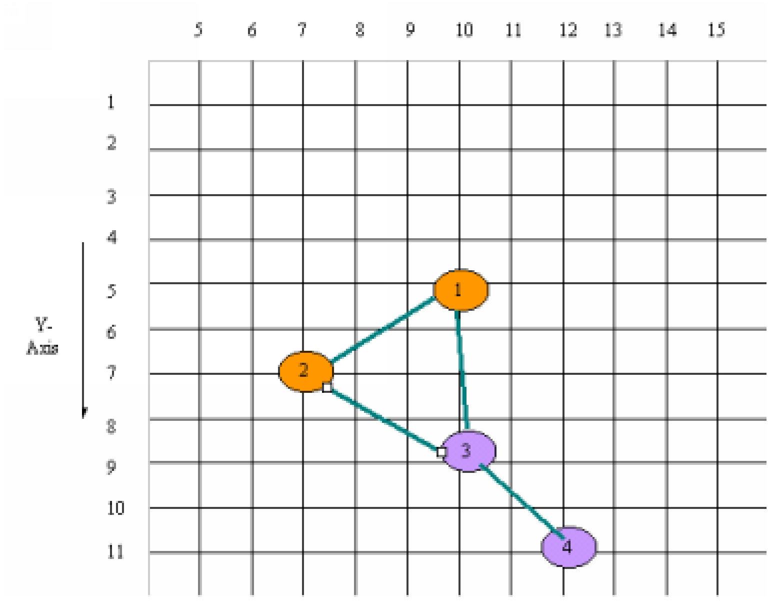


Figure 5.11: Example 4: The actual testbed

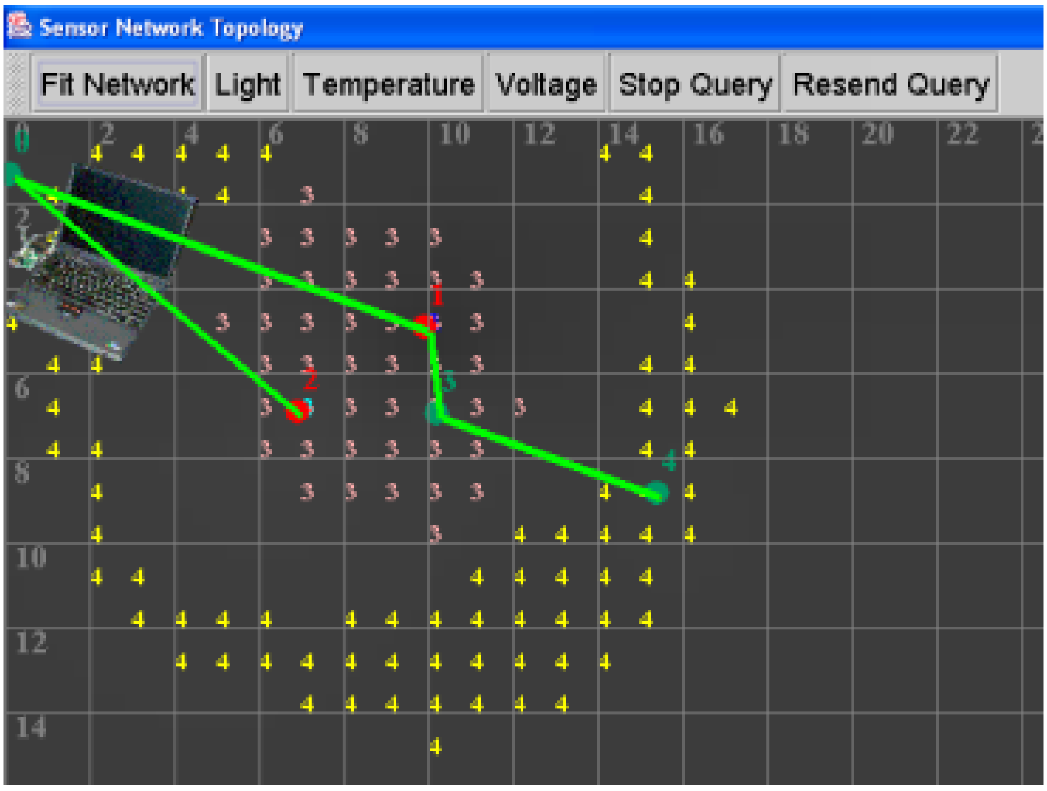


Figure 5.12: Example 4: Output

Chapter 6

Conclusion

Many wireless sensor network applications depend on nodes being able to accurately determine their locations. This is an attempt to study and implement in-range method of localization in a dynamic environment. This method relies only on the sensors knowledge of their neighbors and the transmission range. Our main result is that the in-range technique can provide accurate localization even when memory limits are severe, the seed density is low, and network transmissions are highly irregular. The strengths of this method lie in its simplicity and efficiency where each sensor localizes itself based on the positions of the beacons and also collaborates with other sensors aiding their localization. Many issues remain to be explored in future work including how well our assumptions hold in different mobile sensor network applications, how different types of motion affect localization, and how our technique can be extended to the case where RSSI can be used to determine distances between the nodes.

Chapter 7

Limitations and Future Enhancement

While the in-range method is a simple, non-complex method of finding the positions of the sensors, it does not give the accurate position of the sensor. The maximum error in the position is constrained by the transmission range. The worst-case scenario occurs, when a node is in the vicinity of only a single other node (or beacon), then the possible location of the former node becomes the entire transmission range of the latter.
Secondly, it is not possible to fix the transmission range exactly to a certain number of units. The range oscillates from the fixed value by a small value that can be taken to be negligible but it coarsens the results.
The range changes from environment to environment, from indoor to outdoor. Thus, fixing the range value for the localization algorithm becomes difficult.
The neighbor information from each of the motes is routed to the PC. The localization algorithm can function only it obtains all the neighbor information. This latency is totally dependent on the number of nodes. As the number of nodes is increased, the contention and routing time increases.
As the number of nodes increases, the number of iterations in the localization in-range method will also increase. Hence, the IR range method has a complexity proportional to the number of nodes.

Thus, the next task is to find a more optimized solution to the localization aspect in sensor networks. Finding the approximate distance between the nodes by using other methods could reduce the error in locations. One example is acoustic ranging method, where the distances are reckoned by sending a sound signal and the time of the signal. Thus, the possible locations confines to a hollow band (taking tolerance into consideration) rather than an entire circular area of transmission.
Once the positions of the nodes have been discovered, the next task is to place a buzzer in the room. The sensors are to be placed on moving vehicles (toy cars for e.g.), which periodically sense the buzzer. Based on the intensity of the buzzer signal and the momentary positions of the sensor, the approximate location of the buzzer has to be determined. In this way the localization algorithm could used to track other objects in the environment

Appendix A

nesC Files : Detection of Neighbors

A.1 Query for Neighbors

A.1.1 SendQueryC.nc

```
configuration CmdReg
{

}

implementation {

        components Main, GetNeighbhorsC, SingleTimer, CmdRegM, Command;

        Main.StdControl -> GetNeighbhorsC.StdControl;
        Main.StdControl -> CmdRegM.StdControl;
        Main.StdControl -> SingleTimer.StdControl;
        CmdRegM.Commands -> Command;
        CmdRegM.Timer -> SingleTimer.Timer;

}
```

A.1.1.1 SendQueryM.nc

```
#define MAX_NODE_ID 50
module CmdRegM {

        provides interface StdControl;

        uses {

                interface CommandUse as Commands;
                interface Timer;
        }

}

implementation {

        command result_t StdControl.init() {
                return SUCCESS;

        }
        command result_t StdControl.start() {

                return call Timer.start(TIMER_ONE_SHOT, 1000);
        }

        command result_t StdControl.stop() {

                return SUCCESS;

        }

        event result_t Timer.fired() {

                ParamVals paramVals;
                char resultBuf[10];
                SchemaErrorNo errorNo;
                static uint16_t daddr=1;

                /* Set Destination Address */

                paramVals.numParams = 1;
                daddr = TOS_BCAST_ADDR;
                paramVals.paramDataPtr[0] = (char *)&daddr;

                if(call Commands.invoke("SENDMSG", resultBuf,
                &errorNo, &paramVals) == FAIL)
                        return FAIL;

                return SUCCESS;

        }


        event result_t Commands.commandDone(char *commandName,
        char *resultBuf, SchemaErrorNo errorNo) {

                return SUCCESS;
        }
}
```

A.2 Respond to Request/Neighbor Information

A.2.1 Receiver.h

```
#ifndef GLOBAL
#define GLOBAL
enum {
        AM_GETNEIGHBHORS = 120
};
enum {
        DATA_LEN = 2
};
enum {
        REQUEST = 1
};
enum {
        RESPONSE = 2
};
enum {
        MAX_NUM_OF_NEIGHBHORS = 25
};
enum {
        REFRESH = 255
};

typedef struct NeighbhorData {
        uint8_t saddr;
        uint8_t msgType;
} NeighbhorData;

typedef struct neighbhors {
        uint8_t addr;
        uint16_t rssi;
}neighbhors;
#endif
```

A.2.2 ReceiverC.nc

```
includes GetNeighbhors;

configuration GetNeighbhorsC {
        provides interface StdControl;
}

implementation {

        StdControl = GetNeighbhorsM.StdControl;
        //Main.StdControl -> GetNeighbhorsM.StdControl;
        GetNeighbhorsM.SendMsg -> Comm.SendMsg[AM_GETNEIGHBHORS];
        GetNeighbhorsM.ReceiveMsg -> Comm.ReceiveMsg[AM_GETNEIGHBHORS];
        GetNeighbhorsM.Register -> Command.Cmd[unique("Command")];
        GetNeighbhorsM.Leds -> LedsC;
        GetNeighbhorsM.CommControl -> Comm.Control;
        GetNeighbhorsM.SetValue -> GlobalM.SetValue;
        GetNeighbhorsM.GetValue -> GlobalM.GetValue;
        GetNeighbhorsM.Commands -> Command;
        GetNeighbhorsM.Random -> RandomLFSR;
        GetNeighbhorsM.SetReqSentFlag -> ReqSentFlagM.SetReqSentFlag;
        GetNeighbhorsM.GetReqSentFlag -> ReqSentFlagM.GetReqSentFlag;
        GetNeighbhorsM.RadioControl -> CC1000ControlM.CC1000Control;
        GetNeighbhorsM.MicroTimer -> MicroTimerM.MicroTimer;
        GetNeighbhorsM.RefreshNeighbhorsTable -> GlobalM.StdControl;
}
```

A.2.3 ReceiverM.nc

```
includes GetNeighbhors;

module GetNeighbhorsM {

        provides {

                interface StdControl;
        }

        uses {
                interface CC1000Control as RadioControl;
                interface CommandRegister as Register;
                interface ReceiveMsg;
                interface SendMsg;
                interface StdControl as CommControl;
                interface Random;
                interface Leds;
                interface GetValue;
                interface SetValue;
                interface CommandUse as Commands;
                interface SetReqSentFlag;
                interface GetReqSentFlag;
                interface MicroTimer;
                interface StdControl as RefreshNeighbhorsTable;
        }
}

implementation {

        TOS_Msg retBuf, respBuf, globalTosMsg;
        uint16_t globalDaddr;
        TOS_MsgPtr pmsg=&retBuf;
        NeighbhorData *data;

        command result_t StdControl.init() {

                call CommControl.init();
                call Leds.init();
                call Random.init();

                return SUCCESS;
        }

        command result_t StdControl.start() {

                ParamList paramList;

                call CommControl.start();

                /* Set Radio Power */

                if(call RadioControl.SetRFPower(1) == FAIL) {

                        call Leds.yellowToggle();
                }

                /* Register Command to Send Request */

                paramList.params[0] = UINT16;
                paramList.numParams = 1;
                if(call Register.registerCommand("SENDMSG", VOID, 0,
                  &paramList) == FAIL) {
                        return FAIL;
                }

                /* Set Timer */

                if(TOS_LOCAL_ADDRESS != 0) {

                        call MicroTimer.start(5000000);
                }
                return SUCCESS;
        }

        command result_t StdControl.stop() {

                call CommControl.stop();

                return SUCCESS;
        }

        event result_t Register.commandFunc(char *commandName, char *resultBuf,
                       SchemaErrorNo *errorNo, ParamVals *params) {

                NeighbhorData *PayLoad;
                uint8_t localNodeid=0;

                /* Get Local node ID */

                localNodeid = TOS_LOCAL_ADDRESS;

                /* Assign to Global variables */

                        globalDaddr =
                        *((uint16_t *)params->paramDataPtr[0]);

                /* Fill Payload */

                memset((char *)&globalTosMsg, 0, sizeof(globalTosMsg));
                PayLoad = (NeighbhorData *)globalTosMsg.data;
                PayLoad->msgType = REQUEST;
                PayLoad->saddr = localNodeid;
                globalTosMsg.length = DATA_LEN;

                /* Send Request */

                if(call SendMsg.send(globalDaddr, DATA_LEN,
                &globalTosMsg) == FAIL) {

                        call Leds.yellowToggle();
                }

                call SetReqSentFlag.set(1);

                return SUCCESS;
        }

        event result_t SendMsg.sendDone(TOS_MsgPtr msg, result_t success) {

                char buf[10];
                SchemaErrorNo errNo;

                call Leds.redToggle();

                return SUCCESS;
        }

        event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msg) {
```

```
                TOS_Msg tosMsg;
                TOS_MsgPtr ret;
                NeighbhorData *resData;
                uint8_t localNodeid;
                uint16_t backoffTime;

                /* Buffer swapping */

                call Leds.greenToggle();

                ret = pmsg;
                pmsg = msg;
                memset((char *)&tosMsg, 0, sizeof(tosMsg));
                data = (NeighbhorData *)msg->data;

                /* Process Request */

                if(data->msgType == REQUEST) {

                        /* Get Local node ID */

                        localNodeid = TOS_LOCAL_ADDRESS;
                        resData = (NeighbhorData *)respBuf.data;
                        resData->msgType = RESPONSE;
                        resData->saddr = localNodeid;

                        /* Send Response */

                        if(TOS_LOCAL_ADDRESS == 0) {
                                return ret;
                        }

                        if(call SendMsg.send(data->saddr,
                        DATA_LEN, &respBuf) == FAIL) {

                                call Leds.yellowToggle();
                        }

                        return ret;
                }

                if(data->msgType == RESPONSE) {
                        call SetValue.set(data->saddr, msg->strength);
                }

                return ret;
        }

        event result_t Commands.commandDone(char *commandName,
        char *resultBuf, SchemaErrorNo errorNo) {

                return SUCCESS;
        }

        async event result_t MicroTimer.fired() {

                ParamVals paramVals;
                char buf[10];
                SchemaErrorNo errNo;
                uint16_t daddr;
                static uint8_t refreshCounter=0;

                /* Periodic Refresh of Neighbhors Table once in a Minute */

                refreshCounter++;
                        refreshCounter = 0;
                        call RefreshNeighbhorsTable.start();
                        call SetValue.set(REFRESH, REFRESH);

            /* Send Request for Neighbhors */

                daddr = TOS_BCAST_ADDR;
                paramVals.numParams = 1;
                paramVals.paramDataPtr[0] = (char *)&daddr;
                if(call Commands.invoke("SENDMSG",
                buf, &errNo, &paramVals) == FAIL) {
                        call Leds.yellowToggle();
                }

                return SUCCESS;

        }

} /* End implementation */
```

Appendix B

Java File : Localization.java

```
%\ begin
%verbatim}

/* The following code carries out the localization on the sensors:
The neighbours of the motes are taken as the input to the module and the
iterative procedure is carried out periodically*/

package net.tinyos.tinydb;
import java.io.*;
import net.tinyos.tinydb.global;

public class localization
{
  int noofnodes = 20;
/*no. of nodes is taken to be 20 ; it can be changed accordingly*/

  public static boolean array[][][];
/*the array variable signifies whether it is possible for a node
to be occupied at a particular position
if array[i][x][y]=true => (x,y) is a viable position of node i;
if array[i][x][y]=false => (x,y) is a not a viable position of node i;*/

  boolean nbrs[][];
/* Similarly if nbrs[i][j]=true =>node i is a neighbour of node j;
Similarly if nbrs[i][j]=false =>node i is not a neighbour of node j;*/

  boolean beacon[];
/*if beacon[i]= true  => node i is a beacon mote;
if beacon[i]= false  => node i is not a beacon mote;*/
}

boolean[][]
intersection (boolean arr1[][], boolean arr2[][])
 /*
    calculates the intersection of 2 sets depicting possible locations
  */
{
  boolean result[][] = new boolean[200][200];

  for (int i = 0; i < 200; i++)
    for (int j = 0; j < 200; j++)
      {
        result[i][j] = arr1[i][j] && arr2[i][j];
      }
  return result;
}



boolean[][]
     union (boolean arr1[][], boolean arr2[][]) /* calculates
                                                   the union of 2 sets depicting poss
{
  boolean result[][] = new boolean[200][200];

  for (int i = 0; i < 200; i++)
    for (int j = 0; j < 200; j++)
      {
        result[i][j] = arr1[i][j] || arr2[i][j];
      }
  return result;
}



boolean[][]subrange (int r, boolean arr1[][])
/* subtracts from the set of possible locations of a mote,
 sets of locations which arise from
 the fact that some beacons are not its neighbours */
{
  boolean result[][] = new boolean[200][200];

  for (int i2 = 0; i2 < 200; i2++)
    for (int j2 = 0; j2 < 200; j2++)
      result[i2][j2] = true;

  for (int i = 0; i < 200; i++)
    for (int j = 0; j < 200; j++)
      if (arr1[i][j])
        {
          for (int i1 = i - r; i1 <= i + r; i1++)
            for (int j1 = j - r; j1 <= j + r; j1++)
              if (i1 >= 0 && i1 < 200 && j1 >= 0 && j1 < 200)
                {
                  if ((i1 - i) * (i1 - i) + (j1 - j) * (j1 - j) <= r * r)
                    result[i1][j1] = false;
                }
        }
  return result;
}



boolean[][]addrange (int r, boolean arr1[][])
/* computes the set of locations which are in transmission range
 of a point for each point in a set of possible locations of a mote */
{
  boolean result[][] = new boolean[200][200];

  for (int i = 0; i < 200; i++)
    for (int j = 0; j < 200; j++)
      if (arr1[i][j])
        {
          for (int i1 = i - r; i1 <= i + r; i1++)
            for (int j1 = j - r; j1 <= j + r; j1++)
              if ((i1 - i) * (i1 - i) + (j1 - j) * (j1 - j) <= r * r)
                if (i1 >= 0 && i1 < 200 && j1 >= 0 && j1 < 200)
                  result[i1][j1] = true;
        }
  return result;
}


void
initialize ()                    /* identifies the beacon motes
                                    and sets their positions */
{

  for (int i = 0; i < noofnodes; i++)
    try
    {
      File prim = new File (i + "locX.dat");
      FileInputStream fos = new FileInputStream (prim);
      DataInputStream dos = new DataInputStream (fos);
      int x = (int) dos.readLong ();
      dos.close ();
      fos.close ();
      prim = new File (i + "locY.dat");
      fos = new FileInputStream (prim);
      dos = new DataInputStream (fos);
      int y = (int) dos.readLong ();
      dos.close ();
      fos.close ();
      beacon[i] = true;

      for (int x1 = 0; x1 < 200; x1++)
        for (int y1 = 0; y1 < 200; y1++)
          array[i][x1][y1] = false;

      array[i][x][y] = true;
    }

  catch (Exception e)
  {
    for (int x1 = 0; x1 < 200; x1++)
      for (int y1 = 0; y1 < 200; y1++)
        array[i][x1][y1] = true;
  }

}



void
iterate ()                       /* carries out localisation where a
                                    mote's position is localised to a set of positions
                                    based on the locations of its neighbours */
{
  int noofiter = 5;

  for (int iter = 0; iter < noofiter; iter++)
    {
      for (int node = 0; node < 20; node++)
        {
          boolean y[][] = new boolean[200][200];
          boolean temp[][] = new boolean[200][200];

          for (int m = 0; m < 200; m++)
            for (int n = 0; n < 200; n++)
              {
                y[m][n] = temp[m][n] = true;
              }
          for (int nbr = 0; nbr < 20; nbr++)
            {
              if (nbrs[node][nbr])
                temp = addrange (5, array[nbr]);
              y = intersection (y, temp);
            }
          array[node] = intersection (array[node], y);

        }
    }

  for (int node = 0; node < 20; node++)
    for (int nbr = 0; nbr < 20; nbr++)
      {
        if (!nbrs[node][nbr] && beacon[nbr] && node != nbr)
          {
            boolean temp[][] = new boolean[200][200];
            temp = subrange (5, array[nbr]);
            array[node] = intersection (array[node], temp);
          }
      }
}

/*This function is called when the
first object of the class is created.*/
public void
start1 ()
{
  array = new boolean[20][200][200];
  beacon = new boolean[20];
  nbrs = global.nbrs;
  initialize ();
  iterate ();
}

/* This function is evoked periodically, to refresh
the computation of locations*/
public void
start ()
{
  beacon = new boolean[20];
  nbrs = global.nbrs;
  initialize ();
  iterate ();
}

}
```

References

[1] A. Karnik and A. Kumar. Iterative Localization in Wireless Ad Hoc Sensor Networks: One-Dimensional Case. Research supported by a grant from the Indo-French Centre for the Promotion of Advanced Research (IFCPAR) (Project No. 2900-IT). Institute of Science, Bangalore, 2004.

[2] I.F. Akyildiz et al. Wireless Sensor Networks : A Survey. Georgia Institute of Technology, Atlanta, Computer Networks 38(2002) 393-422.

[3] X.Ma. Location Related Issues in Mobile Network Systems. University of Minnesota, Minneapolis, 2002. AD HOC Wireless Networking pp. 365 - 382

[4] F.Koushanfar et al. Location Discovery in Ad-Hoc Wireless Sensor Networks.University of California, 2002. AD HOC Wireless Networking pp. 137 - 174.

[5] N. Dankwa An Evaluation of Transmit Power Levels for Node Localization on the Mica2 Sensor Mote. Yale University, New Haven, 2004.

[6] N. Bulusu at al. GPS-less Low Cost Outdoor Localization For Very Small Devices. IEEE Personal Communications, Special Issue on "Smart Spaces and Environments", Vol. 7, No. 5, pp. 28-34, October 2000.

[7] D. Estrin et al. Next Century Challenges: Scalable Coordination in Sensor Networks. Proceedings of the Fifth Annual International Conference on Mobile Computing and Networks (MobiCOM '99), August 1999, Seattle, Washington.

[8] N. Bulusu et al. Scalable Coordination for wireless sensor networks: Self-Configuring Localization Systems. Proceedings of the Sixth International Symposium on Communication Theory and Applications (ISCTA 2001), Ambleside, Lake District,UK, July 2001.

[9] D. Estrin et al. Instrumenting the world with wireless sensor networks. Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP 2001), Salt Lake City, Utah, May 2001.

[10] V. Raghunathan et al. Energy-aware wireless microsensor networks. IEEE Signal Processing Magazine, Volume: 19 Issue: 2 , March 2002 Page(s): 40 50.

[11] L. Bajaj et al. GloMoSim: A Scalable Network Simulation Environment. UCLA Computer Science Department Technical Report 990027, May 1999.

[12] S. Madden et al. TinyDB : In-Network Query Processing in TinyOS, 2003.

[13] D. Gay et al. nesC 1.1 Language Reference Manual. 2003.

[14] TinyOS Documentation. http://www.tinyos.net/tinyos-1.x/doc/index.html, 2003.

[15] The Network Simulator - NS2. http://www.isi.edu/nsnam/ns/.

[16] Crossbow - Motes and Smart Dust Sensors in Wireless Sensor Networks. http://www.xbow.com/Products/Wireless_Sensor_Networks.htm.

[17] SensorSim - A Simulation Framework for Sensor Networks. http://nesl.ee.ucla.edu/projects/sensorsim/.