

**A TEMPORAL MESSAGE ORDERING AND OBJECT  
TRACKING APPLICATION**

**by**

**LAKSHMAN KAVETI**

**B.E., Deccan College of Engineering and Technology (affiliated with  
Osmania University), India, 2006**

**A REPORT**

**Submitted in partial fulfillment of the requirements for the degree**

**MASTER OF SCIENCE**

**Department of Computing and Information Sciences  
College of Engineering**

**Kansas State University  
Manhattan, Kansas**

**2008**

**Approved by:**

**Major Professor  
Dr. Gurdip Singh**

## Abstract

TinyOS is an operating system designed for wireless embedded sensor network which supports the component based development language called Nesc. Wireless sensor network are becoming increasingly popular and are being used in various applications including surveillance applications related to object tracking. Wireless sensor devices called motes can generate an event in the network whenever there is some object moving in its vicinity. This project aims to develop an application which detects the path information of object moving in the sensor field by capturing the order of events occurs in the network.

This application builds a logical topology called DAG (Directed acyclic graph) between the motes in the network which is similar to the tree topology where a child can have multiple parents which are in communication range and a level closer to the root. Using a DAG, motes can communicate efficiently to order the events occurring in the sensor field. The root of the DAG is the base station which receives all the events occurred in the network and orders them based on the information it has from previous events received.

Every event occurring in the network is assigned a time stamp and is identified by a tuple (mote\_id, timestamp) which describes that the mote with identity id has detected the object with the timestamp, and ordering all such events based on the timestamps we get the path information. There are two time stamping algorithms written in this project. In the first time stamping algorithm, whenever any event occurs, it updates the timestamp information of the entire neighboring mote in the field and when the object enters in the detection range of neighboring mote of previous detected mote, it assigns the new timestamp. The second time stamping algorithm just send the message to the parent and it passes on to its parent until the message is received at the base station, and base station itself assigns the timestamps based the event on first come first serve basis. The application is tested by displaying the path information received and ordered at the base station.

# Table of Contents

Table of Contents .....	iii
List of Figure.....	vi
Acknowledgement .....	vii
Dedication.....	viii
Chapter 1 - Introduction.....	1
1.1    Wireless Sensor Network.....	1
1.2    TinyOS.....	1
1.3    Applications .....	2
1.4    Limitations .....	2
1.4.1    Limited battery power: .....	3
1.4.2    Message loss due to interference:.....	3
1.4.3    Mote failure: .....	3
1.4.4    Security: .....	3
1.4.5    Storage restrictions:.....	4
1.4.6    No fixed topology: .....	4
1.4.7    Loss because of Obstructions:.....	4
Chapter 2 - Problem Statement .....	5
2.1    Logical topology:.....	5
2.1.1    Assumptions for logical topology: .....	5
2.2    Object Tracking Algorithms:.....	5
2.2.1    Algorithm to maintain the local clocks: .....	6
2.2.2    Algorithm to maintain the clocks at the base station: .....	6
2.2.3    Assumptions for object tracking algorithm: .....	6
2.3    TOSSIM.....	6
2.3.1    TinyViz:.....	7
2.3.1.1    Setting the communication range: .....	7
2.3.2    Tython:.....	8
2.3.2.1    Object movement in TinyViz:.....	8
2.3.2.2    Setting the ADC values: .....	9

2.3.3	Testing on Motes: .....	9
2.3.3.1	Reducing the communication range: .....	9
2.3.3.2	Receive Signal Strength Intensity:.....	10
Chapter 3 - Logical Topology .....		12
3.1	Introduction to DAG:.....	12
3.1.1	Why a DAG is necessary: .....	12
3.2	Formation of the DAG:.....	13
3.2.1	Assumptions:.....	13
3.3	Pseudo Code of DAG formation.....	16
3.4	Performance Analysis .....	21
3.4.1	Analysis using the Simulator.....	21
3.4.2	Analysis when implemented on testbed .....	26
Chapter 4 - Algorithm to Maintain Local Clocks .....		27
4.1	Introduction.....	27
4.2	Assumptions.....	27
4.3	Ordering detected events.....	28
4.4	Pseudo Code.....	30
Chapter 5 - Algorithm to Maintain Clocks at the Base Station.....		33
5.1	Introduction.....	33
5.2	Assumptions.....	33
5.3	Ordering detected events.....	33
5.4	Pseudo Code.....	36
Chapter 6 - Performance Analysis of Object Tracking Algorithms .....		39
6.1	Algorithm to maintain local clocks.....	39
6.1.1	In simulator .....	39
6.1.2	On testbed.....	42
6.2	Algorithm to Maintain Clocks at the Base Station .....	43
6.2.1	In Simulator.....	43
6.2.2	On testbed.....	44
6.3	Compare the Object Tracking Algorithm.....	45
6.4	Wiring of Components used in Object Tracking Application.....	46

References..... 48

## List of Figures

Figure 2.1 – TinyViz .....	7
Figure 3.1 - Motes in DAG structure with Mote 0 as base station. ....	12
Figure 3.2- Object moving adjacent to leaf mote. ....	12
Figure 3.3 - Motes in network where base station is mote 0. ....	15
Figure 3.4 - If needChild message from Mote 1 and Mote 2 is lost due to interference. ....	15
Figure 3.5- needChild message from mote 3 and mote 4 is lost due to collision. ....	16
Figure 3.6 - the resultant DAG. ....	16
Figure 3.7 -Motes arranged in grid of 8X8, Mote 0 is base station which initiates the algorithm. .....	22
Figure 3.8 - When Mote 12 is moved away from the base station. ....	23
Figure 3.9 - Mote 13 also moves away from base station .....	24
Figure 3.10 - Mote 13 moves back to its previous position.....	25
Figure 3.11 – Mote 12 moves back to its initial position.....	25
Figure 4.1 – When object moves adjacent to leaf motes .....	29
Figure 5.1- Logical topology used in the algorithm. ....	34
Figure 5.2 - When object moves adjacent to leaf motes. ....	35
Figure 6.1 – When two objects are moving in the sensor field at same time. ....	40
Figure 6.2 – When object moves adjacent to leaf motes in a network with 24 motes.....	41
Figure 6.3 – When object moves adjacent to leaf motes in a network with 24 motes.....	41
Figure 6.4 – Wiring of components used in Object Tracking Application. ....	47

## **Acknowledgement**

It is pleasure to thank people to who made this report possible.

Firstly, I thank my Major Professor Prof. Gurdip Singh for his continued support and guidance throughout this project. I thank you for your encouragement, sound advices, and great effort to explain things clearly.

I thank my committee members, Prof. Daniel Andersen and Prof. Mitchell L. Neilsen for their support.

Finally, I wish to thank my family and friends for their support, helping me to in difficult times and caring they provided.

## **Dedication**

This report is dedicated to my mother and father, who inspired me and gave me everything I needed.



# Chapter 1 - Introduction

## 1.1 Wireless Sensor Network

Wireless Sensor Network (WSN) is a distributed network of devices in which devices communicate using wireless transmission. WSN have a wide range of applications such as monitoring environmental changes, monitoring animal movements, traffic control, fire detection and surveillance. There are different types of sensors used in different applications. Wireless Sensor Networks consists of wireless devices called motes which can send and receive wireless messages, and read data from the sensors. To develop an application, we need to design a distributed algorithm and embed it into all the motes in the network. The TinyOS operating systems specifically designed for motes, is used in our implementation to execute the developed applications and algorithms. Sensor network usually consists of a multi hop algorithm which passes the messages between motes.

## 1.2 TinyOS

TinyOS is the operating system designed for embedded sensor networks. TinyOS supports Nesc language which is a structured component based language. Supporting libraries and applications of TinyOS are written in Nesc. Nesc follows the C syntax, and it is easy to build the components for the applications, debug using the simulators and allows different concurrency models. Components in Nesc can provide and use the multiple interfaces which are bidirectional. If an interface has commands and the component provides that interface then the component should implement commands of the provided interface in the implementation section, and if it uses any interface, then the component can call the commands and receive events if any declared in the interface. For example, if a component provides the Stdcontrol interface then it must implement `init()`, `start()`, and `stop()` commands. Similarly if component uses the ADC control then it can

call `ADC.getData()` command and in return it gets the event `ADC.dataReady()` which is implemented in the component.

### **1.3 Applications**

Wireless sensor networks can be used for various applications. For example, temperature sensor can be used to detect fire, proximity sensors for animal monitoring and object tracking, light sensor to monitor environmental changes, and accelerometer sensor to find the speed at which a vehicle is moving. While developing an application many aspects needs to be considered such as message loss, mobility of motes, crashing of motes, and dynamic topology changes. In most of the applications, a logical topology is needed to build a robust application as the physical positions and number of motes in unknown since motes can be moved or can crash. We can form different types of logical topologies such as star, mesh, ring, or a tree topology. As the communication range may be limited, an application might need a multi hopping algorithm. For example, consider a situation where there are several producers and one consumer in the network and all of producers are not in the communication range of the consumer. If producers want to send any information which the consumer is interested in, then a producer can send the information another mote closer to the consumer in network which pass on to the message to the consumer. To do this, we need to form a logical topology set up in the network. That is if tree topology is built between the motes and consumer is a root of the tree then information can be propagated to the consumer by sending it via parents in the tree. In the object tracking application studies in this report, we setup a logical topology which is a DAG (Direct Acyclic Graph). This is similar to a tree where a child can communicate to all the parents which are in communication range. Chapter 3 describes why the Directed Acyclic Graph is necessary and how it solves the problem.

### **1.4 Limitations**

There are various sensor networks applications developed which can be deployed in the

places such as forests, battle field and remote places where monitoring of the motes is difficult. This section discusses the limitations which need to be considered while developing application for the sensor networks.

#### **1.4.1 Limited battery power:**

Motes run on the batteries and every message received, sent and processed consumes battery power. Sending and receiving consumes more power than processing. When the battery of a mote gets drained, it will stop responding to the other motes in the network. For example, if there is a root in the tree topology which receives the temperature reading of all motes in the network deployed in forest and sends it to some other devices, then the root will be doing more work than anyone else in the network. So, root consumes more power and may exhaust its battery power before others. Due to remote deployment, replacing batteries in these devices may be difficult. Hence, we must reduce the amount of battery consumption by careful design of the algorithms.

#### **1.4.2 Message loss due to interference:**

Messages may get lost due to collisions when there is more than one message transmitted at the same time in a neighborhood. . So, the application developer needs to address this issue when designing an algorithm. If there is a message sent, the sender cannot assume that the messages will be received at the other end.

#### **1.4.3 Mote failure:**

In application deployed in places such as forests or remote places, preventing physical loss of motes in such places is a difficult task. For example, some animal can step on a mote which will damage the mote. So, applications must be designed to continue working even if some motes in the network stop working.

#### **1.4.4 Security:**

When any sensitive information is passed between the motes, the security

of this information needs to be considered in the network. To ensure security, a message needs to be encrypted before it is sent and should be decrypted after it is received. Encrypting the message will increase its size and the amount of computation which will consume more battery power.

#### **1.4.5 Storage restrictions:**

The motes do not have much storage space in it. So, it is not preferable to send large messages, and store large amount of data in the motes of network.

#### **1.4.6 No fixed topology:**

As the motes can crash because of power consumption or physical loss, the topology between motes cannot be fixed. So, an application should work even when there are mote failures. This can be address by having the logical topology created between motes periodically, but it increases in the number of messages sent in network.

#### **1.4.7 Loss because of Obstructions:**

If there are any objects in between the two motes which are trying to communicate, then the signal strength can be reduced or the signal can become weak so that the motes will not receive the messages. Depending upon the obstruction, messages may also get lost.

## **Chapter 2 - Problem Statement**

This project aims to develop the application to track the object moving in the sensor field. The project is divided into two parts, first is the setting up of logical topology, and second is the algorithm which runs in the motes to get the path information of the object moving in the sensor field. Each event detected is assigned the timestamp, and all the timestamp information is gathered at the base station where path information can be read. This project can also be used to order events that occurred in the sensor field.

### **2.1 Logical topology:**

In order for the motes to communicate in the sensor field there needs to be some logical topology set up. As motes can be crashed or moved, the logical topology needs to be updated periodically. The algorithm which sets up the logical topology is initiated periodically to setup and update the topology between the motes in the sensor field. The factor of message loss is considered in the algorithm and reliability is maintained by waiting for the acknowledgment from the receiver at the sender.

#### **2.1.1 Assumptions for logical topology:**

- There is dense enough network to set up the logical topology.
- There will be only one base station which instantiate algorithm.

The logical topology which will be setup is called DAG (Directed Acyclic Graph). DAG is similar to the Tree structure where child can communicate to the parents which are in the communication range. It is needed for object tracking algorithm developed. The reason is described in the following section.

### **2.2 Object Tracking Algorithms:**

Two object tracking algorithms are developed.

### **2.2.1 Algorithm to maintain the local clocks:**

This algorithm is developed to track an object in the sensor field. In this algorithm the logical clock are maintained at all the motes in the sensor networks.

### **2.2.2 Algorithm to maintain the clocks at the base station:**

This algorithm is also developed to track the object movement, but the time stamping is done at base station only. That is, all the events occurring in the network are ordered on first come first serve bases at base station.

### **2.2.3 Assumptions for object tracking algorithm:**

- DAG is already formed in between the motes in the network.
- Message move faster than the object move in between the motes which are in the communication range.

Performance analysis is done for the DAG formation algorithm and object tracking algorithms. Comparison between the two object tracking algorithms in the different scenarios is presented.

## **2.3 TOSSIM**

This section describes how the TinyOS simulator is used to write and test an application. We have written two Nesc files for the DAG formation and for ordering the events in the object tracking algorithm. One file is used for specifying the component interconnections and the other for implementing the commands and events of interfaces used by each component in the application. Application can be debugged by including debug statements in the code. The application was developed and tested using Cygwin which is the UNIX simulator for the Windows Operating system.

## 2.3.1 TinyViz:

TinyOS provides the graphical user interface called TinyViz. TinyViz provides features such as setting the communication range between two motes, debugging the application by using the debug messages used in the code and aligning the motes in the network as needed to test the application.

To test the application, we need to compile the application which can be done using the “make pc” command. Before running the application, we need to set the DBG variable using the command “export DBG=usr1” to print the debugging statements written in the code. Using “tinyviz -run build/pc/main.exe 16” command, the application can be launched in TinyViz (the parameter 16 specifies the number of motes).

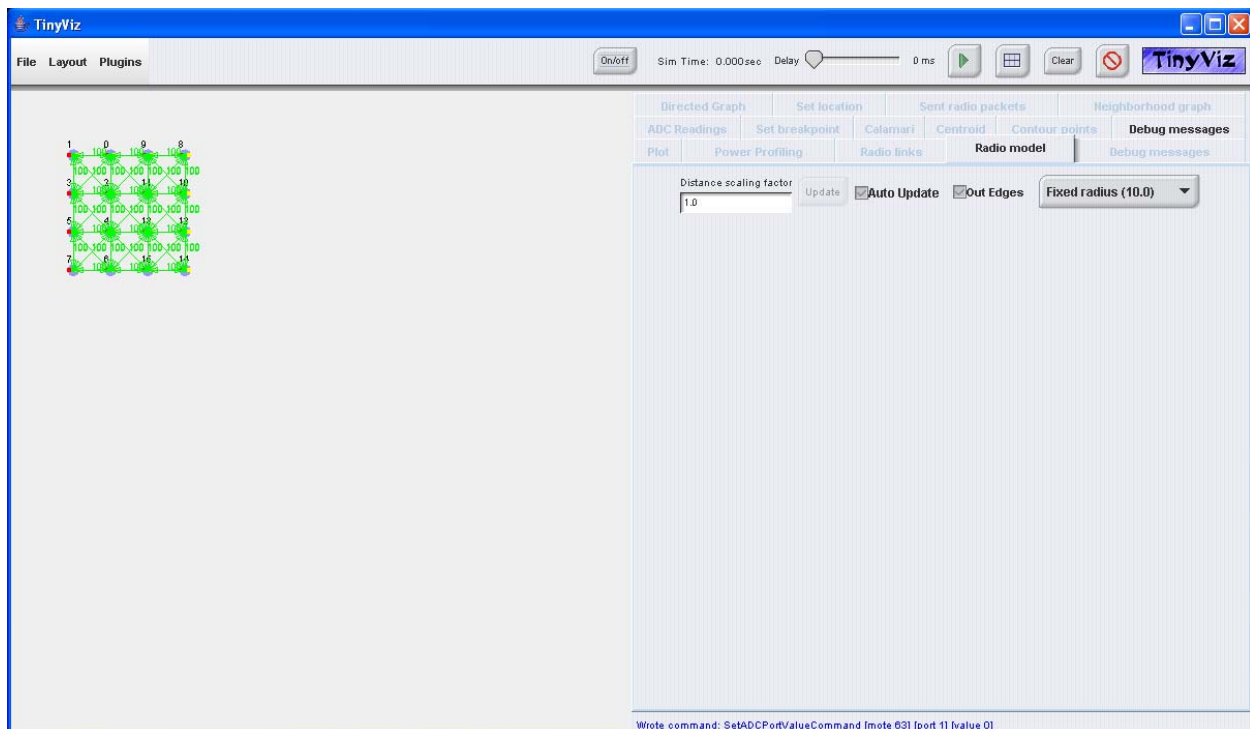


Figure 2.1 - TinyViz

### 2.3.1.1 Setting the communication range:

The application was tested by setting the communication range between motes in network and creating the multi hop network in TinyViz. By

clicking on the Plugins and selecting the Radio model checkbox, the tab for the radio model appears. Here, by selecting the fixed radius (10.0) in drop down list and clicking the Auto Update check box, TinyViz sets the communication distance between any two motes to 10 units. Now, we can make two motes communicate by placing them within a distance of 10 units in the user interface. .

### 2.3.2 Tython:

Tython (Tinython) adds a scripting interface to TOSSIM. User can interact with the TinyViz application using Tython scripting. Our application uses Tython to simulate object movement in TinyViz and setting the ADC values of the motes which are at the 5 units of distance from the object. When the application is loaded in TinyViz, it shows up the scripting prompt “>>>” where Tython commands can be given to the simulator.

#### 2.3.2.1 Object movement in TinyViz:

A new instance of an object can be created and moved in the simulator using the Tython script. First we need to run the following command in the Tython prompt to start the simdriver.

```
>>> from simcore import *
```

Now the motes instances can be accessed using the Motes[] array. Motes[0] give the instance of the mote with id 0. And the command Motes[0].moveTo(x,y) moves mote 0 to the coordinates (x,y) in the sensor area of TinyViz. Using this feature, we can set the exact topology needed by arranging all the motes. We can have a new object created in the sensor area of TinyViz by running the script below

```
>>> obj=sim.newSimObject(1,10,20)
```

Where 1 is the size and, 10 and 20 are the x and y coordinates respectively of the new object created. Similar to the moving of existing mote, the new object created can also be moved in the TinyViz interface.



### **2.3.2.2 Setting the ADC values:**

Similar to the object movement, setting the ADC values of the mote can also be done using the Tython script. It can be done by running the script below

```
>>> comm.setADCValue(10,0,1,5);
```

Where 10 is the mote id, 0 is the time, 1 is the port and 5 is the ADC values. This will set the ADC values of mote 10 to 5. We tested the application by setting the ADC values of a mote to 5 if the distance between object and mote is less than or equal to 5 and setting it to 0 when the distance is greater than 5 units. Objects are moved by the distance of 3 units every second using the `obj.moveTo(x,y)` command.

### **2.3.3 Testing on Motes:**

The testing is done on the test-bed, consisting of a set of boards, where each is of height three feet, and one foot width. Each board has eight motes on it in a grid of 2X4, two rows and four columns. The effective range of a TelosB mote in an outdoor environment is 75 to 100m and indoor is 20 to 30m. With this communication range, it is difficult to test in a small area. So, the experiments are performed by decreasing the transmission power to reduce the communication range between motes.

#### **2.3.3.1 Reducing the communication range:**

While working with the motes in a small area, we can reduce the effective communication range of motes using the CC2420Control interface. CC2420Control interface has the command `SetRFPower(uint8_t power)`, where power takes the values 1 to 31. If power is set to 31, it uses the full transmission power to send the message and messages reach the maximum distance it can. This project is tested by setting the transmission

power to 2 as below in the start command of StdControl interface.  
call CC2420Control.SetRFPower(2);

### 2.3.3.2 Receive Signal Strength Intensity:

When forming a DAG, sometimes a mote may receive a weak signal and get an incorrect level. If a mote gets a level number by receiving a weak signal, it cannot effectively communicate later. This affects the event ordering in the Object Tracking algorithm. For example, mote X may get level 2 by receiving a weak signal from one of the motes which is at level 1, but the mote X is supposed to be in the level 3 as the distance between mote X and the root node is 3. So, when mote X detects the object, it send the detected message to level 1 motes, but as the distance to level 1 motes is more than expected, the signal might not reach it parents and hence the detected message might not reach the base station.

To solve such problems, Receive Signal Strength Intensity (RSSI) is used while forming the DAG. We can calculate the RSSI of Telosb using the formula below.

```
(int8_t)recv_packet->strength-45
```

Where `recv_packet` is `TOS_MsgPtr` (TinyOS message pointer). The strength field is in the `TOS_Msg` structure. The structure of the `TOS_Msg` for the Telosb is

```
typedef struct TOS_Msg
{
    /* The following fields are transmitted/received on the radio. */
    uint8_t length;
    uint8_t fcfhi;
    uint8_t fcflo;
    uint8_t dsn;
    uint16_t destpan;
    uint16_t addr;
```

```

uint8_t type;
uint8_t group;
uint8_t data[TOSH_DATA_LENGTH];

/* The following fields are not actually transmitted or received
 * on the radio! They are used for internal accounting only.
 * The reason they are in this structure is that the AM interface
 * requires them to be part of the TOS_Msg that is passed to
 * send/receive operations.
 */
uint8_t strength;
uint8_t lqi;
bool crc;
bool ack;
uint16_t time;
} __attribute__((packed)) TOS_Msg;

```

In this structure the first 10 bytes are reserved for the header information and last 5 values are not set or transmitted, but are used for internal accounting. In the last 5 fields we have the signal strength which gives the intensity of the signal strength when received at a mote. The range of signal strength is from -1 to -100dBm. In this project, messages received with the signals strength of -85dBm or below is considered as weak and messages are processed only when the message received has the strength of more than -85dBm while forming the DAG.

# Chapter 3 - Logical Topology

## 3.1 Introduction to DAG:

As mentioned in the previous chapters, we form a DAG (Directed acyclic graph) between the motes in the network. A DAG is similar to the Tree structure where a child can have more than one parent. In this topology, all motes enrolled in the DAG have the level number assigned to them and motes which are in communication range and level one less than it own are the parents and the motes with the level one greater are the children. Below figure describes the DAG structure.

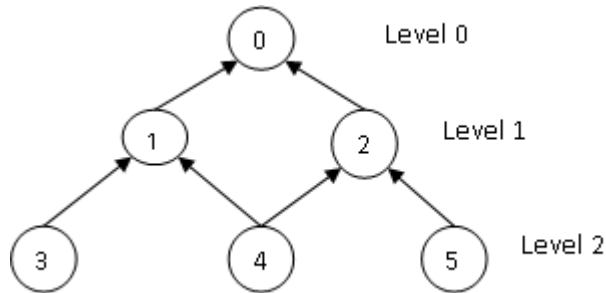


Figure 3.1 - Motes in DAG structure with Mote 0 as base station.

### 3.1.1 Why a DAG is necessary:

The Object tracking algorithm needs a DAG for ordering the detection events and updating the local clocks. For example consider the scenario below

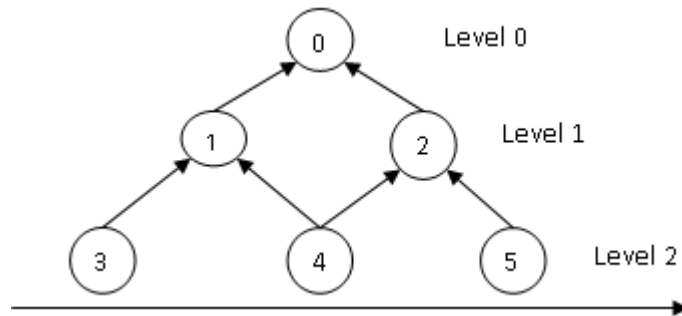


Figure 3.2 - Object moving adjacent to leaf mote.

As show in the figure above, if the object moves adjacent to the leaf motes,

then mote 3 detects the object first and assigns the timestamp 0 to the detection event, and if mote 4 is not in the communication range then it is not possible for mote 3 to update the local timestamp of mote 4. So, mote 1, which is the parent of mote 3 and mote 4, updates the timestamp of mote 4. Similarly, before mote 5 detects the object, its local timestamp should also be updated which has to be done by the common parent of mote 4 and mote 5. Since, mote 4 needs a common parent for both the siblings; it needs at least two parents, one for each sibling, which is possible by using a DAG structure. Hence, this project uses logical topology of DAG between the motes in the network for ordering the detection events fired in the sensor field.

## **3.2 Formation of the DAG:**

An algorithm is written to form a DAG between the motes in the sensor field. The base station initiates the algorithm which forms the DAG with the base station as the root of the DAG. There is only one base station which initiates the algorithm. Motes do not know the neighboring motes information initially. The algorithm is designed to know the information of neighboring motes by sending the messages and getting the acknowledgment from them.

### **3.2.1 Assumptions:**

- We assume that we have a dense enough network to form the DAG.
- There will be only one base station which initiates the algorithm.

#### **Basic idea to form DAG:**

When the base station starts the algorithm, it broadcasts the needChild message. Whoever receives the needChild message will acknowledge it by sending the message ackToParent and adds that motes which sent the needChild message as a parent. When the ackToParent message is received at the base station, it will add the mote which sent the

ackToParent message as its child and acknowledges it by sending an ackToChild message. When ackToParent is received by a child, it waits for the sometime to allow messages from other potential parent to arrive, and then sends a needChild message and starts searching for more levels by sending the needChild message.

But there can be a situation where we can form an incorrect DAG .For example, if the ackToParent message is lost due to interference, and then the parent does not get to know the child's information. For this reason, we ensure reliability by sending the acknowledgments to all the parents which sent the needChild message every one second unless they get the ackToChild message back from all the parents. But, if the needChild message itself is lost then there is no information to track the children of that mote.

### **Mobility of motes and changing topology:**

Mobility is also considered when developing the DAG formation algorithm. When a mote receives the needChild message from a mote which is at a level closer to the base station than its parents, then it changes its level to a lower value and makes the mote which sent the needChild message as parent. Similarly, if the object moves farther from the base station then it gets the needChild message from the siblings or the children but not from the parents, when algorithm is initiated again. If this situation occurs the algorithm resets the level to the default which is -1. When base station initiates the algorithm again, it gets the right level, parents and children in the DAG. Like this when the object moves in the sensor field then the level is updated accordingly using this algorithm. Since the topology might change as the mote may crash or move, this algorithm has to be run at periodic intervals. Hence, we make it a heartbeat-based algorithm.

Consider the network below where base station is mote 0

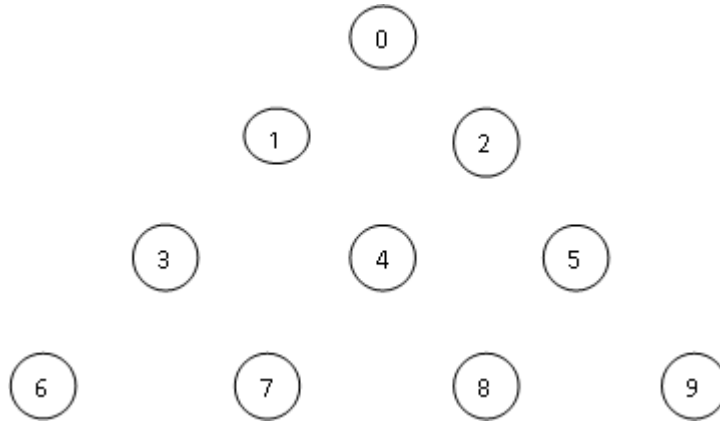


Figure 3.3 - Motes in network where base station is mote 0.

When the base station initiates the algorithm, mote 1 and mote 2 gets into level 1. Algorithm needs to make all the motes which are in the communication range other than the parents of the node as the children. For example, in the network below, if mote 1 and mote 2 send the 'needChild' message at the same time then due to interference, messages may be lost and all other motes does not gets enrolled into the DAG.

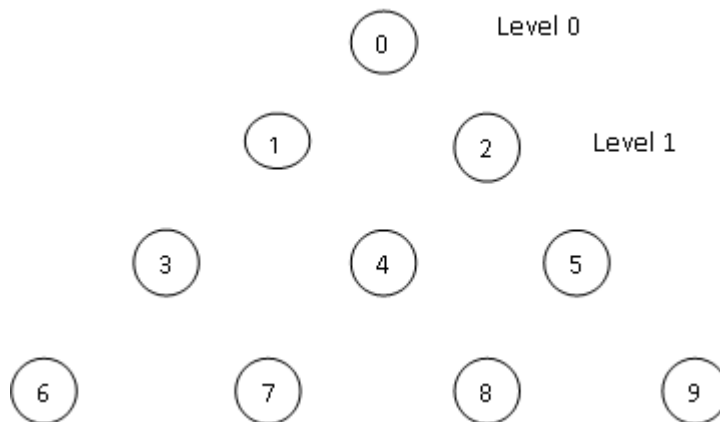


Figure 3.4 - If needChild message from Mote 1 and Mote 2 is lost due to interference.

Another situation may be if mote 3 and 4 get the needChild message correctly and are enrolled in the tree with level 2. Level 2 motes will now send 'needChild' message to form the DAG with the motes which are away from the base station. If the mote 3 and mote 4 send messages at the same time then because of interference in the medium, the messages sent from mote 3 and mote 4 will be lost. But, messages sent from mote 5 are received at mote 8 and 9, and the mote 8 and 9 go into the level 3. Similarly, mote 7 goes

to level 4 and mote 6 goes to level 5 if the mote 7 is in communication range of the mote 8 and mote 6 is in the communication range of mote 7. Like this we get a wrong level for the motes.

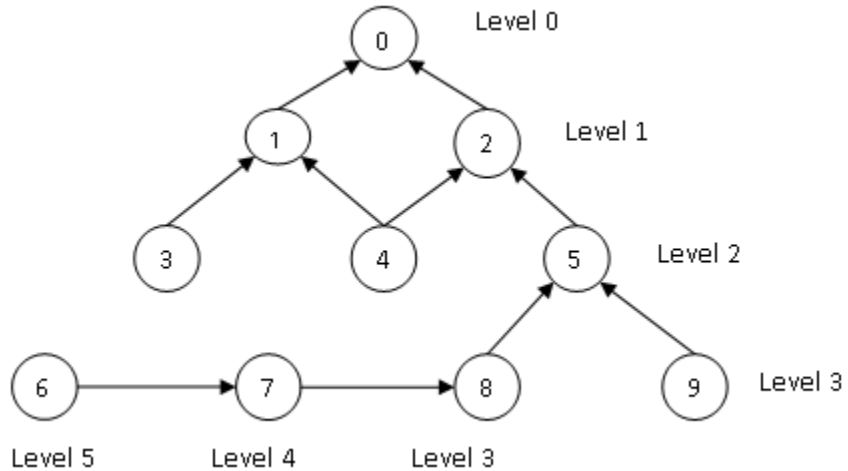


Figure 3.5 - needChild message from mote 3 and mote 4 is lost due to collision.

To solve such problems we make it a heartbeat algorithm. Base station initiates the algorithm periodically. Mote 6 and Mote 7 might get the needChild message from mote 3 and mote 4, and changes the level of mote 6 and mote 7 to 3. The resultant DAG would be as below

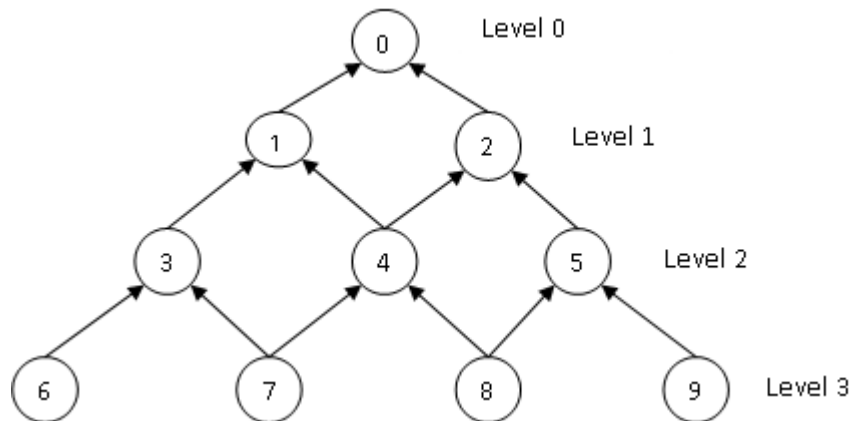


Figure 3.6 - the resultant DAG

### 3.3 Pseudo Code of DAG formation

Pseudo code for DAG formation is written to explain the algorithm in detail. Promela style is used in Pseudo Code.



```

int baseStation = 0;
int ackFrmParent[10], noOfAck_pnt = 0;
int parents[10], childs[10];
bool notInitiated, inRightLevel = FALSE;
Timer timer1, timer2, timer3, timer4, timer5, timer6;
do
    Local_ID = baseStation && notInitiated -> notInitiated = false; (bcast)!
    msgStruct(msgStruct->message="needChild"); timer1_start; timer5_start;
    levelno = 0;
    ?msgStruct ->
        if
            msgStruct->message == "needChild" ->
                if
                    levelno == -1 || levelno == msgStruct->level + 1 →
                        inRightLevel = TRUE;
                if
                    ( (msgStruct-ID not in ackFrmParent[]) && timer2_called
                    == 0 ) ->
                        ackFrmParent[noOfAck_pnt] = msgStruct->ID;
                        timer2_start; noOfAck_pnt++; levelno =
                        msgStruct->levelno+1; ( msgStruct->ID include in
                        parents[] );
                    ( (msgStruct_ID not in ackFrmParent[]) && timer2_called
                    == 1 ) ->
                        ackFrmParent[noOfAck_pnt] = msgStruct->ID;
                        noOfAck_pnt++; levelno = msgStruct-
                        >levelno+1; ( msgStruct->ID include in parents[] );
                fi
            levelno > msgStruct->levelno + 1 → inRightLevel = TRUE;
            if
                ( (msgStruct-ID not in ackFrmParent[]) &&

```

```

timer2_called == 0 ) ->
    ackFrmParent[noOfAck_pnt] = msgStruct-
    >ID; timer2_start; noOfAck_pnt++; levelno
    = msgStruct->levelno+1; ( msgStruct->ID
    include in parents[] );
( (msgStruct_ID not in ackFrmParent[]) &&
timer2_called == 1 ) ->
    ackFrmParent[noOfAck_pnt] = msgStruct ->
    ID; noOfAck_pnt++; levelno = msgStruct ->
    levelno+1; ( msgStruct->ID include in
    parents[] );
fi
if
    levelno < msgStruct->levelno + 1 → timer6_start;
fi
fi
msgStruct->message == "ackToParent" ->
    ( msgStruct->ID include in childs[]); !msgStruct
    (msgStruct->message = "ackToChild" );
msgStruct->message == "ackToChild" && notInitiated ->
    (msgStruct->ID remove from ackFrmParent[] );
    timer3_start;
msgStruct->message == "ackToChild" && !notInitiated ->
    ( msgStruct->ID remove from ackFrmParent[] );
fi
!notInitiated -> timer4_start;
timer1_fired -> notInitiated = true, inRightLevel = FALSE;
timer2_fired ->
    if
        (ackFrmParent[] is notEmpty) ->
            (get nodeX from ackFrmParent[]); (!msgStruct(

```

```

                                msgStruct->message = "ackToParent") to nodeX);
                                (ackFrmParent[] is Empty) -> timer2_stop;
                                fi
timer3_fired ->
                                notInitiated = false; (bcast)!msgStruct(msgStruct ->
                                message="needChild"); timer1_start;
timer4_fired-> clear all the variable;
timer5_fired->
if
                                notInitiated->
                                    (bcast)!msgStruct(msgStruct->message="needChild");
                                    timer1_start;
                                fi
timer6_fired->
if
                                !inRightLevel && !baseStation -> level = -1;
                                fi
od

```

A reason for placing timeout is described below:

**Timer1:**

Timer1 is REPEAT timer placed for the base station and this just changes notInitiated to true so that it can initiate the formation of DAG again. It is the repeat timer but in the implementation the timeout time has to change every time by adding more 60 seconds. That is, if first time it timeouts after 60 seconds then second time it timeouts after 120 seconds and 180 seconds for the third time, and so on.

**Timer2:**

Timer2 is the REPEAT timer which ticks every second. The purpose of placing

this is to ensure reliability of communication between parent and child after receiving the message from the parent mote. The algorithm remembers the motes which sends the needChild messages and which are not the child's of the nodes by placing the ID's in the ackFrmParent array. When timer2 fires it check if there are any ID's of nodes in the ackFrmParent array, if there are any then send the "ackToParent" to nodes with ID's in ackFrmParent array until it becomes empty. When the array is empty then stop the timer2. Array removes its values when it gets the "ackToChild" from the parent. Therefore it makes sure that the parent has accepted it as child and messages are successfully passed between parents and children.

**Timer3:**

Timer3 is the ONE\_SHOT timer when a mote is accepted as a child. When this timer is fired it sends the needChild message to motes which are away from the base station than the present level motes. The purpose of keeping this timer and sending the message when timer is fired is to wait for some time so that the nodes can accept more parents and send needChild message only once. Doing this will save some message been sent in the medium.

**Timer4:**

Timer4 is used to clean all the variables used while forming DAG like notInitiated, ackFrmParents and any variables used in the implementation. Assuming formation of DAG process does not take more than 60 seconds, in implementation this timer fires after 60 seconds so that it can treat the next received needChild message as a new message.

**Timer5:**

Timer5 is a REPEAT timer initiated at the base station which will start the algorithm again. This repeat timer will fire with the increased interval of time if the number of child does not increase. That is, if the number of children of base station is same and previously it fired after 60 seconds then it fires after 120

seconds next time. Similarly, if the number of children increases then timer fires by decreasing the time by 60 seconds next time, as the topology is changed.

**Timer6:**

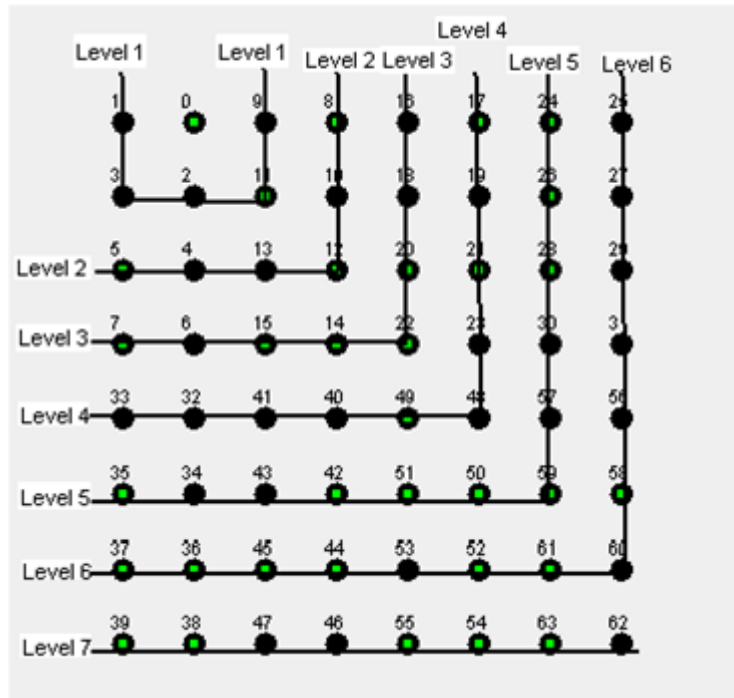
Timer6 is the ONE\_SHOT timer and is initiated when there is any message received from the mote at the level higher (That is, mote farther to the base station). This is used to find whether mote is gone to the wrong level by catching the weak signal or by moving farther from base station.

### **3.4 Performance Analysis**

Various tests were performed on this algorithm to analyze the performance of the algorithm with varying topology sizes in the simulation using TinyViz and also on the motes.

#### **3.4.1 Analysis using the Simulator**

Using the simulator it is easy to find the number of messages sent and arrange the desired topology. In simulator there will be no loss of messages due to the signal strength. So, the desired topology is formed first time itself.



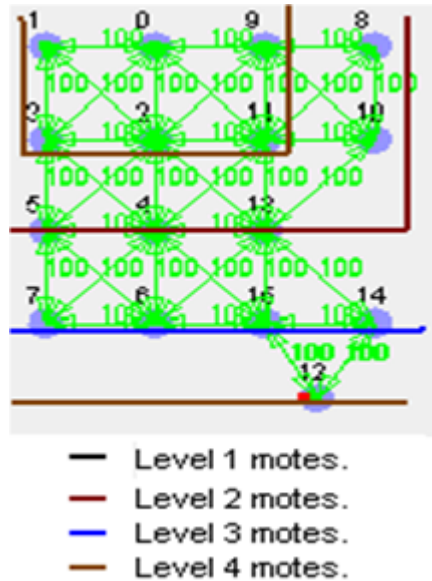
*Figure 3.7 - Motes arranged in grid of 8X8, Mote 0 is base station which initiates the algorithm.*

The DAG formation algorithm is tested with different topologies and network sizes. The average number of messages sent is 71 in the network with eight motes arranged in the grid of 4X2, that is with 4 rows and 2 motes in each row. Similarly, the average number of messages sent is 156 in network of 16 motes arranged in 4X4 grid, 154 in network of 16 motes arranged in 8X2 grid, 253 in network of 24 motes arranged in 4X6 grid, and 965 in network of 64 motes arranged in 8X8 grid. Above figure is the snapshot when the DAG is tested with sixty four motes with base station as mote 0.

If we are trying to form a DAG (Directed acyclic graph) topology with 8 motes then there is a less probability of the message loss due to the interference. The probability of getting interference increases as the number of motes in the network increases as there will be more messages moving in network at same time. As we ensure reliability while formation of tree in algorithm, if messages are lost due to collisions, the messages

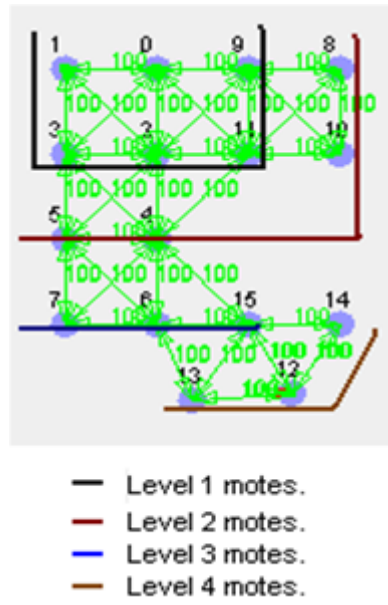
will be resent. This, results in increasing the number of messages sent in the network.

Tests are also performed by moving the objects in TinyViz and results observed are described here. Starting with the topology with 16 motes in a grid of 4X4, the application is tested by moving the motes to observe the algorithm behavior.



*Figure 3.8 - When Mote 12 is moved away from the base station.*

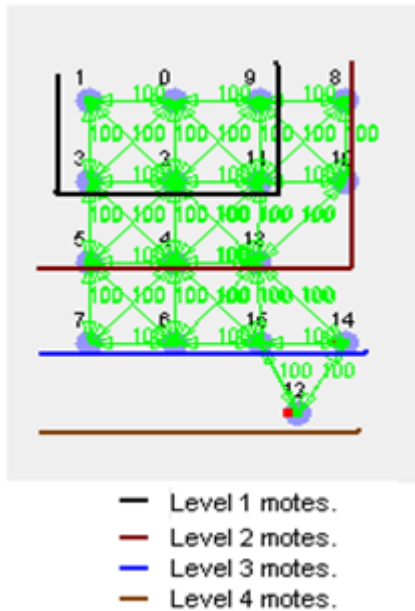
When the algorithm runs for the first time, mote 13 and mote 12 are the parents of the mote 15 and mote 14. As shown in the above figure, mote 12 moves away from base station. When the algorithm is initiated next time, the mote 12 went to the level 4 and, mote 15 and mote 14 became the parent of the mote 12. The parent, child and level information of mote 15 and mote 14 are not changed as there is still one parent (mote 13) in communication range with mote 15 and mote 14. It simulates the behavior where one parent of the mote 14 and mote 15 (I.e., mote 12) is failed, and new mote comes in to network below the mote 15 and mote 14.



*Figure 3.9 - Mote 13 also moves away from base station*

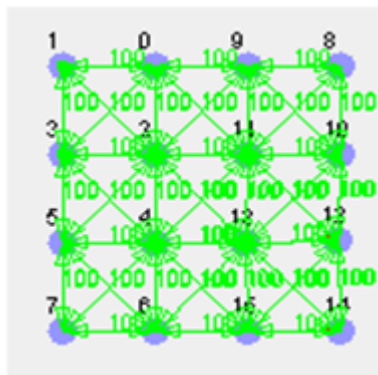
As shown in the above figure, if mote 13 moves away from the base station. When algorithm runs next time the mote 14 went to the level 4 as all the motes in the level 3 are not in communication range now, and mote 15 became its parent. Mote 13 also went to the level 4 and mote 6 and mote 15 became its parent.





*Figure 3.10 - Mote 13 moves back to its previous position.*

As shown in the above figure, mote 13 moves back to initial place. The level of mote 13 is changed to 2 as the mote 2 and mote 11 which are at the level 1 can now communicate, and after receiving the needchild message from mote 2 and mote 11 it changes level to 2 and accepts the mote 2 and mote 11 as the parents. Now, mote 14 can communicate to the mote 13 which is at level 2. So, mote 14 changes the level to next level of the mote 13, which is level 3. And the level on mote 12 remains the same.



*Figure 3.11 - Mote 12 moves back to its initial position.*

When mote 12 is moved back to the same place where it was previously as shown in the figure above. All the motes have the levels, parents and child

as it was initially.

### **3.4.2 Analysis when implemented on testbed**

We tested the algorithm on the testbed which has 8 telosb motes on it. DAG formation algorithm is tested with the 16 motes by joining two testbeds. Sometime, the expected DAG is form for the first time itself, even if it is not formed first time, correct DAG is formed when the algorithm is ran for two or three times.

## Chapter 4 - Algorithm to Maintain Local Clocks

This project developed two algorithms to order the events detecting movement of object moving in the sensor field. The first algorithm maintains the local clocks and second algorithm maintains the clocks at the base stations only. This chapter discusses the algorithm which maintains the local clocks.

### 4.1 Introduction

This algorithm gives the path information of the object moving in the sensor field with a DAG logical topology set up between the motes in the network. The path information is collected at the base station. In this algorithm, whenever any mote in the network detects the object moving in its vicinity, it fires an event and assigns the timestamp for that event. Initially, all the motes will have the timestamp set to 0 and they update it based on the messages they receive. When any mote detects the object, it updates the timestamps of the motes in its communication range. The next section describes in detail how the algorithm orders the event with the following assumptions.

### 4.2 Assumptions

- The DAG logical topology is already set up between the motes in the network.
- Messages moves faster than the object.

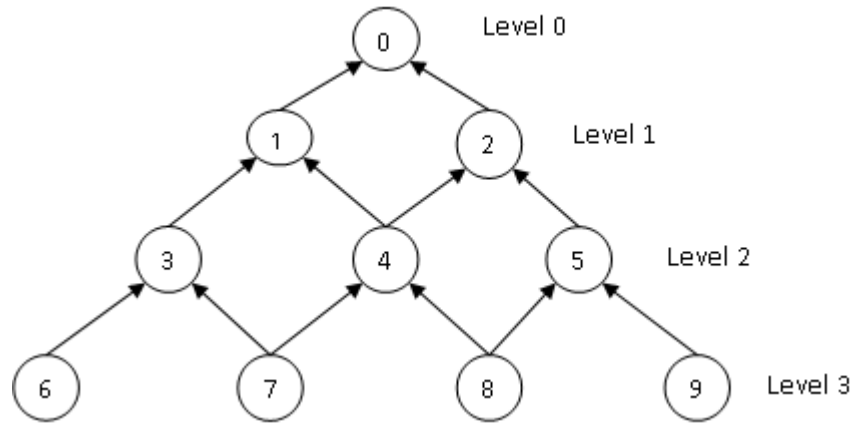
In real time, the communication range will be around 70m to 100m and detection range is around 10 feet. As, motes take only millisecond to send or receive messages the second assumption is a valid assumption.

### 4.3 Ordering detected events

Algorithm follows the below steps to order the detected events.

- Initially all motes will have the timestamps set to zero.
- Path information is maintained at all the motes in the form of tuples (mote\_id, timestamp). For example,  $\{\{1, 0\}, \{2, 1\}, \{3, 2\}\}$  means that mote 1 detected the object with timestamp 0, 2 with timestamp 1, and 3 with timestamp 2.
- Whenever any mote detects the object in the proximity, it will broadcast the detected message with local address, level of the mote detected object and timestamp information to all the neighboring nodes which are in communication range. It updates the path information using the data it has and increment its local timestamp by 1.
- When any mote receives the detected message from its child, it will check whether it already has the path and timestamp information. If it has then it just ignores else it updates the path and timestamp information and passes on to its neighbors and updates the path and timestamp information of its parents, siblings in communication range and child's.
- When a detected message is received from the parent, it will check whether it already has that path and timestamp information. If it has then it just ignores else it updates the (path, timestamp) information.
- When a mote receives the message, it will update its timestamp to the maximum of its timestamp and timestamp in the received message incremented by one.

Using these steps we can order the detected events. Consider a scenario where object moves adjacent to the leaf motes as shown in below figure.



*Figure 4.1 - When object moves adjacent to leaf motes*

When mote 6 detects the object it updates the local path information and broadcasts the detected message. Mote 3 which is the parent of mote 6 receives the message and updates the local path and timestamp information, and broadcasts the detected messages. The detected message forwarded by mote 3 is received by mote 7 which is the sibling of mote 6 and mote 1 which is the parent of mote 3, and updates the timestamp and path information. Similarly, the message is passed on till it reaches the base station.

If mote 6 and mote 7 are in communication range then the timestamp of mote 7 is updated when the mote 6 broadcasts the detected message (mote 6, timestamp 0) and if object come in the detection range of mote 7 then it injects the (mote 1, timestamp 1) detected message into the network. If mote 6 and mote 7 are not in the communication range, then before the object come in the detection range of the mote 7, mote 3 which is the common parent receives and broadcasts the detected message of mote 6 to mote 7, and mote 7 updates its timestamp to one. This happens because the object moves slower than the messages. Similarly, before the mote 8 detects the object, it gets the detected message and updates its timestamp, and mote 8 injects the detected message with the information (mote 8, timestamp 2). Similarly, mote 9 will detect the object and send the message (mote 9, timestamp 3). All this information is merged at the base station and displayed.

## 4.4 Pseudo Code

In this section Pseudo Code of this algorithm is explained.

Description of variables used is

- timestamp is to maintain the local timestamp of mote.
- Level is to maintain the level number of the mote in the tree.
- Path is 2D array which has the path information along with the timestamp. For example: {{1, 0}, {2, 1}, {3, 2}}; means mote 1 detected the object with the timestamp 0, 2 with the timestamp 1, and 3 with the timestamp 2.
- n is to maintain the number of the motes detected the object (that is the size of the Path array).
- detected is the boolean variable which will be set to true when motes detects any object.
- detectMsg, and recvdMsg are the Structures which has the information about the timestamp, level, and mote id which sent the message.
- timer\_detect is the timer which fires every second. It detects the object in the proximity and sends the detected information to the neighbors.

Pseudo Code described in promela style is

```
int timestamp = 0;
int level;
int Path[][];
int n = 0;
boolean detected = false;
Timer timer_detect;
Struct detectMsg
{
int timestamp;
int level;
int msg_src;
```

```

}detectMsg;
Struct recvdMsg
{
int timestamp;
int level;
int msg_src;
}recvdMsg;
do
    timer_detect(fires every 500ms) →
        if
            Object in proximity → detect = TRUE;
            Object not in proximity → detect = FALSE;
        fi
    detected →
        (detectMsg → timestamp) = timestamp;
        (detectMsg → level) = level;
        (detectMsg → msg_src) = TOS_LOCAL_ADDR;
        (bcast)!detectMsg;
        UpdatePathInfo(TOS_LOCAL_ADDR, timestamp);
        timestamp++;
   ?(recvdMsg) →
        UpdatePathInfo(recvdMsg → msg_src, recvdMsg → timestamp);
        UpdateTimeStamp(recvdMsg → timestamp);
        if
            ((recvdMsg → level) > level) →
                (recvdMsg → level) = level;
                (bcast)!recvdMsg;
            ((recvdMsg → level) <= level) → Skip;
        fi
od
UpdateTimeStamp(int ts)

```

```
{
// Comments: If ts is the received timestamp in the message, then timestamp of
present mote will be ts + 1 when received timestamp is greater or equal to its local
timestamp. And, ignore if timestamp received in the message is lesser than ts.
    if
        ts >= timestamp → timestamp = ts + 1;
        ts < timestamp → skip;
    fi
}
UpdatePathInfo(int mote_addr, int timestamp)
{
// Comments: This accepts mote_addr and timestamp as parameters and merges
with the path information it already has and stores in the path[][] local variable.
}
```



## Chapter 5 - Algorithm to Maintain Clocks at the Base Station

Similar to the algorithm discussed in chapter 4, one more algorithm was developed, where the sensor field is divided into sensor areas, each of which has its own DAG topology set up, and timestamps are maintained and updated at base stations only. This chapter describes the algorithm which maintains clocks at base station in detail.

### 5.1 Introduction

This algorithm give the path information of the object moving in the sensor fields with the DAG logical topology set up between the motes in all the sensor areas. In this algorithm whenever any mote in the network detects the object moving in its vicinity, it fires an event. The path information is generated at the base station when it gets the detected message from the motes in the DAG. The next sections describe in detail how the algorithm orders the event with the following assumptions.

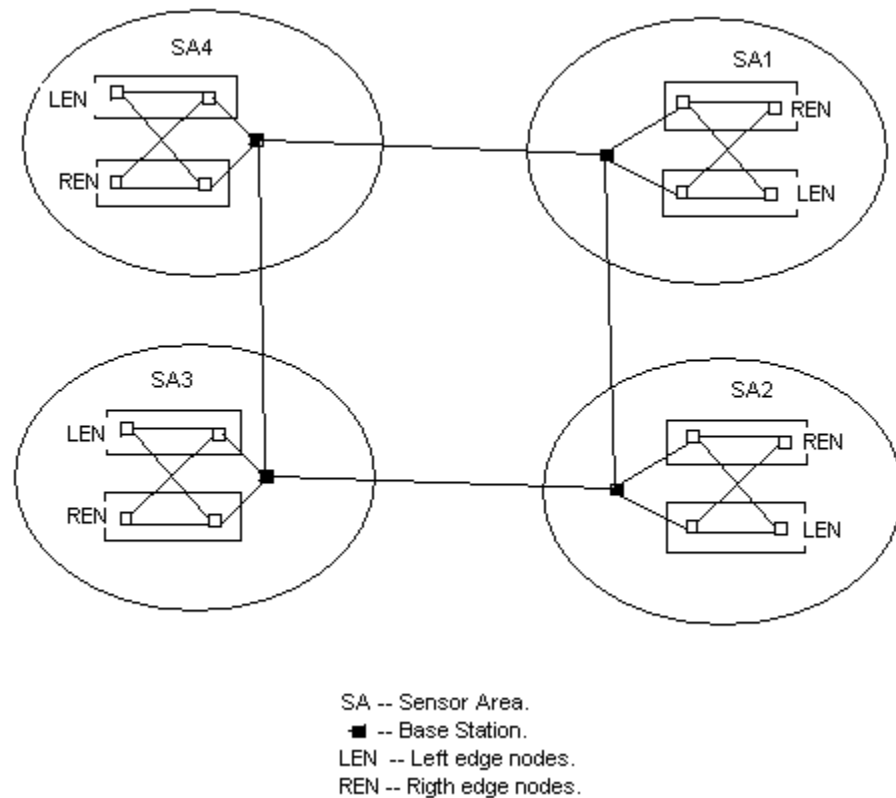
### 5.2 Assumptions

- Sensor field is divided into sensor areas. Each sensor area has logical topology of DAG set up with root as the base station.
- There is a TCP connection between the base stations of each Sensor Area.
- The ring topology is setup between the base stations of Sensor Areas.
- Base station knows the edge information of all the motes in its Sensor Area.
- Messages move faster than the object.

### 5.3 Ordering detected events

In this algorithm, when object moves in the proximity of a mote, it sends the detected event to a randomly chosen parent and the parent which received the messages will pass on to its parent. Similarly, messages are passed on until it

reaches the base station and base station assigns the timestamp on the first come first serve basis. Based on the information base station has, it updates the timestamp information of the neighboring Sensor Area using the TCP connections already established. The figures below describe the topology information and the how the sensor areas are formed in the network.



*Figure 5.1- Logical topology used in the algorithm.*

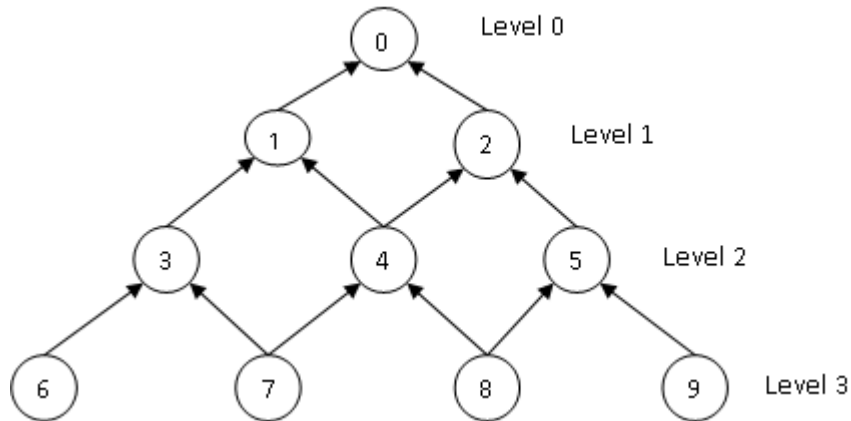
The algorithm follows the step given below to order the events.

- Timestamps are maintained at the base stations only.
- Path information is maintained at the base stations only.
- Whenever any mote detects any object in its proximity, it will send the detected message with its address to the randomly chosen parent. Upon receiving the detected message from the child it will forward to its parent.
- When base station receives the detected message, it updates the path and timestamp information, and based on the edge information of mote which

sent the detected message, it will send the information to the next base station. For example, if base station in the SA1 got the message from the LEN (Left Edge Node), then it will update the timestamp information of the SA2 base station.

- Updating timestamp logic at Base station: Ignore if base station already received the detect message from the mote which sent the message, or else increment the timestamp by 1.
- Updating the Path Information at Base station: Ignore if base station already received the detect message from the mote which sent the message, or else update the Path information by adding the received message source to the Path information it has.

Consider the example used to describe the algorithm which maintains local clocks.



*Figure 5.2 - When object moves adjacent to leaf motes.*

When the object moves the adjacent to the leaf motes as shown in the figure above, mote 6 detects the object first and detected message is sent to mote 3 which passes on to its parent. The same procedure is followed until messages reach the base station. When the base station receives the detected message from mote 6, it updates the path information by adding the tuple (mote 6, timestamp 0) and increases its timestamp to 1. When mote 7 detects the object, it sends a detect message to mote 3 or mote 4 which passes to their parents. When message reaches the base station, the path information and timestamp is updated. Similarly, mote 8

and mote 9 detect messages also reach at the base station. Unlike the algorithm discussed in Chapter 4, here neighboring motes timestamp is not updated when any mote detects the object. So, if the object moves fast in the network then mote 7 message can reach the base station earlier than the mote 6 message and base station would give the path information as (mote 7, timestamp 0), and (mote 6, timestamp 1). As object moves slower than the messages in the network, there is a very rare possibility of such scenarios.

## 5.4 Pseudo Code

In this section Pseudo Code of this algorithm is explained.

Description of variables used is

- timestamp variable is to maintain the local timestamp of mote (this is needed at the base station only).
- level variable is to maintain the level number of the mote in the tree.
- Path is an array which has the path information along with the timestamp (this is needed at the base station only). For example:  $\{\{1, 0\}, \{2, 1\}, \{3, 2\}\}$ ; means mote 1 detected the object with the timestamp 0, 2 with the timestamp 1, and 3 with the timestamp 2.
- n is to maintain the number of motes detected the object (that is, the size of the Path array).
- detected is the boolean variable which will be set to true when motes detects any object.
- detectMsg, and recvdMsg are the Structures which has the information about the level, and mote id which sent the message.
- timer\_detect is the timer which fires every second. It detects the object in the proximity and sends the detected information to the neighbors.

Pseudo Code described in Promela style is

```
int timestamp = 0;
```

```

int level;
int Path[][];
int n = 0;
boolean detected = false;
Timer timer_detect;
Struct detectMsg
{
int level;
int msg_src;
}detectMsg;
Struct recvdMsg
{
int level;
int msg_src;
}recvdMsg;
do
    timer_detect(fires every second) →
    if
        object in proximity → detected = TRUE;
        object not in proximity → detected = FALSE;
    fi
    detected →
        (detectMsg → level) = level; (detectMsg → msg_src) =
        TOS_LOCAL_ADDR; (bcast)!detectMsg;
        if
            (TOS_LOCAL_ADDRESS in BaseStation) →
                UpdateTimeStamp(TOS_LOCAL_ADDRESS,
                timestamp); SendToAppropriateBaseStation(
                TOS_LOCAL_ADDRESS);
            TOS_LOCAL_ADDRESS not in BaseStation → Skip;
        fi
fi

```

```

?(recvdMsg) →
    if
        ((recvdMsg → level) > level) →
            (recvdMsg → level) = level;
            (bcast)!recvdMsg;
        ((recvdMsg → level) <= level) → skip;
    fi
    if
        TOS_LOCAL_ADDRESS in BaseStation →
            UpdateTimeStamp(recvdMsg → msg_src
                timestamp);SendToAppropriateBaseStation(
                recvdMsg → msg_src);
        TOS_LOCAL_ADDRESS not in BaseStation → Skip;
    fi
od
UpdateTimeStamp(int msg_src, int ts)
{
    if
        msg_src not in path[][] → path.add(msg_src, ts); timestamp++;
        msg_src in path[][] → Skip;
    fi
}
SendToAppropriateBaseStation(int msg_src)
{
    if
        msg_src in LEN → Establish TCP connection and send timestamps
            to left Basestation;
        msg_src in REN → Establish TCP connection and send timestamps
            to right Basestation;
    fi
}

```

## Chapter 6 - Performance Analysis of Object Tracking Algorithms

Object tracking algorithms which maintains local clocks and other maintaining clocks at the base station is tested in various scenarios. This chapter describes the interesting scenarios observed while testing the algorithms.

### 6.1 Algorithm to maintain local clocks

Algorithm maintaining the local clocks in explained in the chapter 4. In simulation it is tested with various topologies and moving object in different directions. Algorithm it tested on 2 testbed with 8 telosb motes on each.

#### 6.1.1 In simulator

Different topologies are setup and algorithm is tested by moving the object in various directions. Number of messages sent in the network is more if object moves adjacent to leaf motes than compared to the object moving adjacent to the motes which are closer to the base station. For example, in a network of with 16 motes when object moves adjacent to leaf motes number of messages sent are 23, and when object moves closer to the base station, the number of messages sent is only 6. The number of messages taken is approximately same if object moved towards base station or away from the base station. For example, number of messages sent in a network of 16 motes when object moves towards and away from the base station are 10. There are some more interesting cases observed when multiple objects are moved in the network. That is, if detect events are fired at different places in network then there will are some concurrent detection been reported at the base station.

If two objects are moving in the network at the same time which are far

enough that the detected message can be sent to the motes which have detected the other object, then based on which mote has detected object we can order the detected events although there are chances of getting some concurrent events. But, if the objects are far as shown in the right figure below then there can be more concurrent events happening in the network.

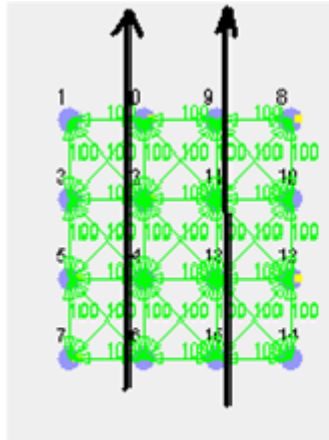


Figure 1

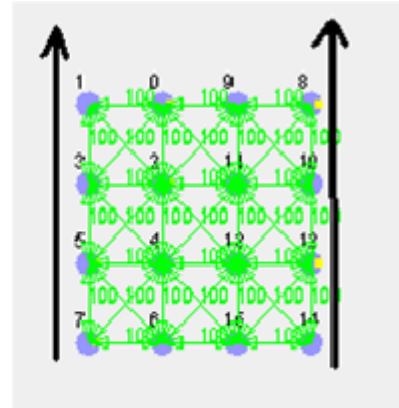


Figure 2

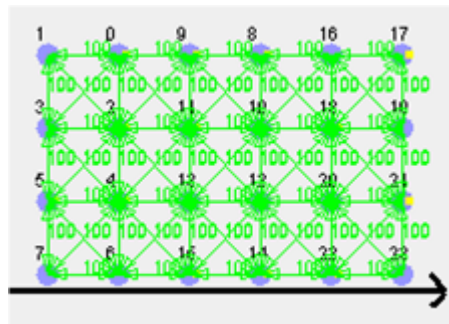
Figure 6.1 - When two objects are moving in the sensor field at same time.

Since, in the DAG each child can have any number of parents, and for this application we assume that there are at least 2 parents for each mote to order the event apart from the motes on edges. If detected message is sent from the child to 2 parent, and those 2 parent in turn sends to 4 motes which are parents of parents (Even if there is one common parent, there can be 3 parents of parents) forwarding the message to base station. So, even if there is any message loss due to collision, some messages will carry information to the base station. But if the detected message itself is lost due to interference, then there is no way to regenerate the information. We could set up the reliability by waiting for acknowledgment from at least one parent, but it increase the messages sent in network which can effect detections of other events as it results in more collisions.

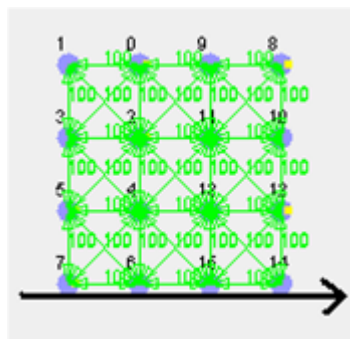
If there is large network, then there will be more number of messages sent



while detecting the motion of the object. Since, if detected message is sent from the child to 2 parents, those 2 parents in turn sends to 3 or 4 motes which are parents of parents forwarding the message to base station. If there is a large network we can expect more levels, parents, and parents of parents which results in sending more messages in network. This scenario is expected if there is a large network. For example, in network with 16 mote when object is moved adjacent to leaf nodes then 23 messages are sent in network, and in network is with 24 motes, 36 messages are sent in network.



*Figure 6.2 - When object moves adjacent to leaf motes in a network with 24 motes.*



*Figure 6.3 - When object moves adjacent to leaf motes in a network with 24 motes.*

If object move horizontally adjacent to the leaf motes then the messages sent in the network are more compared to the object moving adjacent to the motes closer to the base station. And, If object move vertically towards the base station, it takes less number of messages been sent in the network

than the number of messages been sent when the object moves adjacent to the leaf motes.

### **Stress testing of Application:**

With the network of 16 or below mote, by getting adc value at all the motes every second and moving object 2 units every second where distance between the objects is minimum 6 units, we can track the movement of the object with this algorithm correctly. But if we increase the network to 24 motes, then there are more chances of message loss as there will be more object detection events happening and passing in the network. Some times tython is not able to set the adc values of all the motes that much fast. Message loss is relatively more than network with 16 or less motes.

### **6.1.2 On testbed**

On testbed, the algorithm is tested with 8 and 16 motes which have DAG topology setup already and moving the object in different directions. As mentioned in chapter 2, testing on motes is done by reducing the communication range and considering only strong signals using the RSSI values. Base station is connected to computer, and it sends the message to computer whenever it updates the timestamp and path information. A Java application is written to read the data sent from the base station and display on computer.

Results observed when tested on testbed are similar to the simulation results. Number of messages sent in the network is more if object moves the adjacent to leaf motes compared to object moving closer to base station. The number of messages sent in network when object moved adjacent to leaf motes is 29 when tested on one testbed and 48 when tested

on two testbeds. And, when object moved closer to the base station the number of messages sent is 4 when tested on one testbed and 20 when tested on two testbeds. The results also show that the number of messages sent will increase with the network size. Similar the results are taken when object moves towards base station and away from base station. The number of messages sent in the network will be approximately same when object moves towards base station or away from base station. The number of messages sent in network when object moved towards base station is 22 with one testbed and 23 when tested on two testbeds.

## **6.2 Algorithm to Maintain Clocks at the Base Station**

Algorithm to maintain the clocks at base station is explained in the chapter 5. Similar to the algorithm maintaining local clocks this algorithm is also tested in simulation and on testbed.

### **6.2.1 In Simulator**

Similar to the first algorithm, this algorithm is also tested with different topologies and moving the object in various directions. Number of messages sent in the network is more if object moves adjacent to leaf nodes than compared to the object moving adjacent to the nodes which are closer to the base station. For example, in a network of with sixteen nodes when object moves adjacent to leaf nodes number of messages sent are 30, and if object moves closer to the base station only 10 messages are sent. The number of messages taken is same if object moved towards base station or away from the base station. There are some more interesting cases observed when there are multiple objects moving in the network.

If two objects are moving in the network at the same time and speed which are far enough that the detected information can be sent to the node which

has detected the other object, then events are ordered based on which mote detected messages reached the base station. There will be no concurrent events reported by this algorithm as the events are ordered at the base station on first come first serve basis. If two objects are moving, one adjacent to the level 1 motes and other adjacent to the level 4 motes then the ordering done at base station may not be the expected result. Since, before the event which is fired at the level 4 reaches the base station another event may fire at level 1 motes and reach the base station as it needs only one hop. So, level 1 event is ordered first and the level 4 will be second. But the event at the level 4 is fired first than the event fired at the level 1.

In this algorithm, network size does not affect the number of messages sent in the network. Since, the detected message will be sent to any one randomly chosen parent which in turn passes on to any randomly chosen parent.

### **6.2.2 On testbed**

This algorithm is tested on the 3 testbeds with 8 motes on each testbed connected in the ring topology. Whenever any detected message reaches the base station it needs to update the timestamp information of the neighboring base station based on edge information it has. C based application is written which sets up the ring topology between the base stations and receives the messages sent from base stations by establishing the TCP connection and passes it on to the neighboring base station by reading the messages. Each testbed is has unique group id in which they communicate. Motes in one testbed will not communicate with the motes in other testbed as they have different group ids.

The results on the testbed might not be exactly same as what we got in

simulation since when object moves slowly the mote can fire the detected event more than once. This scenario is removed while testing in simulation to get the exact number of messages sent in network when object is detected only once. The results observed when application is tested on testbed are similar to the results in simulation. When object moves adjacent to leaf motes the number of messages sent in the network is more compared to object moving adjacent to motes which are closer to the base station. With one testbed the number of messages sent when object moves adjacent to the leaf motes is 11 and when object moves adjacent to motes which are closer, that is motes at level 1 is 5. The number of messages sent in network when object moves towards base station and away from the base station will be approximately same. On testbed, the number of messages sent when object moves away from the base station is 11 and the number of messages sent in network when object moves towards base station is 13 which is approximately same. This algorithm is also tested with 3 testbed connected in the ring. The number of messages sent in network when the object moves adjacent to the leaf nodes is 31. And the number of messages sent in network when an object move closer to the base station is 14. The number of messages sent in network when object moves towards or away from the base station is approximately same.

### **6.3 Compare the Object Tracking Algorithm**

This section discusses the different scenarios, and describes which algorithm is good for which situation.

- **Clocks:**

In first algorithm clocks and path information are maintained at all the motes. Whenever any detected message is received the message is processed. In second algorithm there is no overhead of maintaining the clocks at the all the motes. The timestamps are assigned at base stations only.

- **Reporting the detected events:**

In Algorithm which maintains the local clocks, the detected event needs to be reported to all the neighbors. But in an algorithm which maintains clocks at base station needs to report the detected event only to one parent.

- **Network Size:**

In algorithm maintaining the local clocks, the number of messages sent in the network increase as the network size increases. In algorithm maintaining the clocks at base station it reports the detected event only to one parent. So, network size does not affect much in ordering the events. For example, in a network with 16 motes, the number of messages sent when object moves adjacent to the leaf motes is 23. With the algorithm to maintaining clocks at the base station it took 16 motes.

- **Interference:**

In algorithm to maintain local clocks, the detected event is sent to all the parents in the communication range and all the parents which received detected message will be sent their parents. And, in algorithm maintaining the clocks at base station detected event will be sent to only one parent and the motes in different sensor area do not interfere as they have different group id's. Hence interference will be more in the algorithm which has local clocks.

## 6.4 Wiring of Components used in Object Tracking Application

Below diagram describes the wiring of the components used in Object Tracking Application. This Component modeling is done in Cadena.

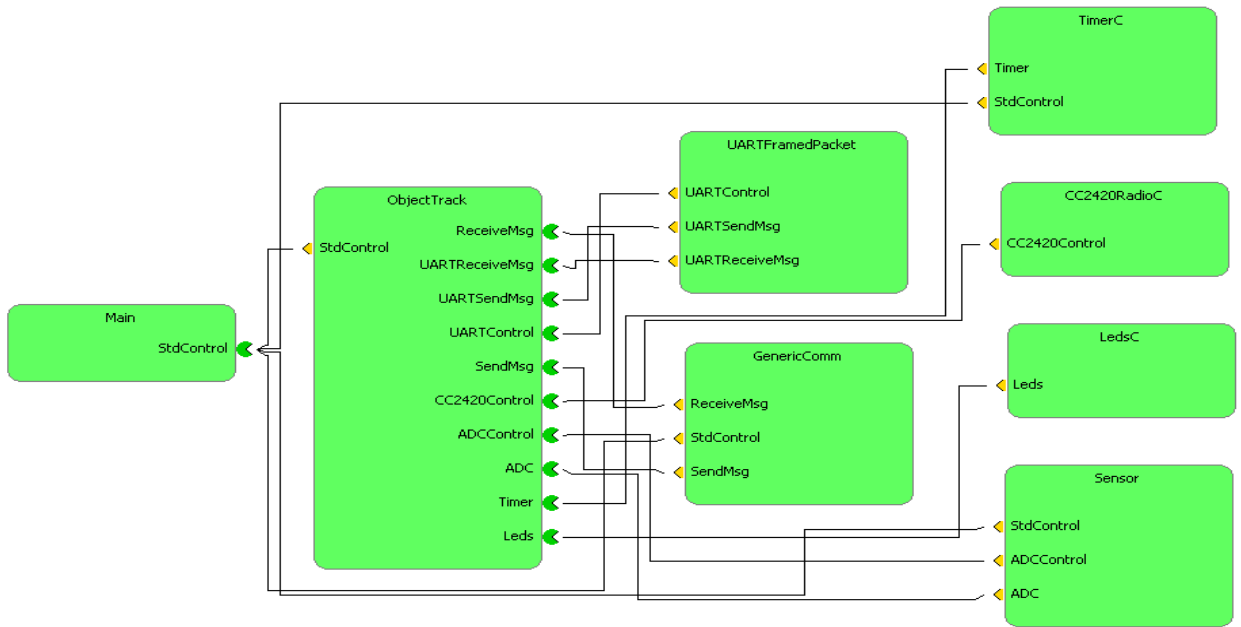


Figure 6.4 – Wiring of components used in Object Tracking Application.

## References

1. TinyOS Tutorial  
<http://www.tinyos.net/tinyos-1.x/doc/tutorial/>
2. Tython Tutorial  
<http://www.tinyos.net/tinyos-1.x/doc/tython/manual.html#sec:intro>
3. Kay Romer: TEMPORAL MESSAGE ORDERING IN WIRELESS SENSOR NETWORKS, Proceedings of the Annual Mediterranean Ad Hoc Networking Workshop 2003.
4. Leslie Lamport: Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM, 21, 1978.