# DESIGN, IMPLEMENTATION AND EXPERIMENTS OF SENSING TASK SCHEDULING IN WIRELESS SENSOR NETWORKS

By

Arunkumar Venkateshwaran

Bachelor of Science

Madras University

Tamil Nadu, India

April 2004

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2008

DESIGN, IMPLEMENTAION AND EXPERIMENTS
OF SENSING TASK SCHEDULING
IN WIRELESS SENSOR NETWORKS

Thesis Approved:

Dr.Xiaolin Li

Thesis Adviser

Dr.Johnson Thomas

Dr.Ning Wang

Dr. A. Gordon Emslie

Dean of the Graduate College

# Table of Contents

# List of Tables

# List of Figures

# Chapter I Introduction

A sensor network consists of a large number of nodes, and each node has a limited amount of energy, memory, and processing and communication capabilities. These constraints make a sensing task to be completed using in-network collaborative processing a prerequisite for most wireless sensor network applications. Quick response in a sensor network results in energy savings; therefore for any wireless sensor network application it is desirable to complete any specific task within a minimum time.

In-network processing is critical to sensor network's lifetime. Since sensor networks are committed to few applications, application-specific code can be distributed through the network and activated when necessary or distributed on-demand. Techniques such as data aggregation can reduce traffic, while collaborative signal processing can reduce traffic and improve sensing quality [1].

A task may be arbitrarily divisible which has the property that all elements in the load require an identical type of processing. Such applications have been called divisible loads because a scheduler may divide the computation among worker processes arbitrarily, both in terms of number of tasks and of task sizes [2]. Sensor nodes gather sensor readings, which can be arbitrarily partioned into any number of load fractions among different sensor nodes. Since these sensor readings do not have any precedence relations each partition of the load can be processed independently and computation based on the data can also be processed independently by a single processor (sensor node). Therefore DLT can be applied to obtain optimal solutions to minimize the finish time in sensor networks.

Divisible Load scheduling theory (DLT) involves the study of optimal distribution of partition able loads among the processors and links in a system. Most of the recent work on DLT provides an efficient algorithm for task allocation to processors in a network by considering the processing time as well as the communication time. DLT does not recognize precedence relations among data; it assumes that computation and communication loads can be partitioned arbitrarily among numerous processors and links, respectively [3]. Therefore our main aim is to obtain an efficient algorithm to partition the load among the sensor nodes in order that the task can be partitioned and processed among the processors in the shortest time.

Various experiments were conducted to sense task allocation using DLT. The preliminary results show that the proposed approach completes any given task in a short time. Since most of the applications require processing of large and frequently occurring data, Huffman encoding algorithm has been used on the nodes to compress the data. Therefore helping to send maximum data in one packet across the network and improving the number of readings sent in one packet and also resulting in saving energy.

## *1.1 General Objectives*

- Provide optimal solutions to minimize the finish time in sensor networks.
- Conducted experiments for task allocation using 2 level Divisible load scheduling theory
- Implemented encoding technique on sensor networks to increase the number of bytes sent in a single packet.
- Conducted experiments on using encoding and decoding techniques on data packets in DLT.
- Soil moisture sensors were integrated to the sensor nodes (tmote) physically as well as programmatically, and effectively stored and retrieved data using the

sensor node's external flash. Experiments were conducted for various issues and deployed in the agriculture field.

## *1.2 Outline of thesis*

The rest of the document is organized as follows: Chapter 2 presents the related work and background knowledge on DLT; Chapter 3 presents the divisible load theory for data fusion, algorithm used, preliminary results and the 2 level Divisible Load theory approached used; Chapter 4 describes problem description, approach, challenges, and setup in a real-time sensor network application for Precision Agriculture project. Chapter 5 has all the concluding notes.

# Chapter II Related Work and Background

## *2.1 Divisible Load Theory*

Scheduling of loads has also been categorized as either job scheduling or task scheduling. A job is defined to be composed of a number of tasks. If a job in its entirety is assigned to a processor, it is called job scheduling, as in distributed computing systems. If different tasks are assigned to different processors, it is called task scheduling, as in parallel processing systems. Thus, the kind of scheduling depends primarily on the type of load being processed.

DLT provides a tractable and practical approach to load scheduling problems involved in sensing, communication and computation aspects. Divisible load scheduling has been an active area of research over the past few years. The authors of [4] provide a collaborative task scheduling algorithm (CTAS) to minimize the event misses and energy consumption by overlapping the sensing area of each node in a cluster.

Extensive research has been studied in the area of energy efficient task allocation and scheduling with discrete dynamic voltage scaling in uni-processor systems. Most of the previous work on energy aware task allocation networks focused on minimizing the overall energy dissipation on the whole network which may not be suitable for wireless sensor networks as each sensor node has its own energy level [5]. The fact that the remaining energy on each sensor node varies was always overlooked in the past. [5], is the first work for task allocation in wireless sensor networks that considers the time and energy costs of both the computation and communication activities. They propose an energy balanced allocation of a real time application on to a single-hop cluster of sensor networks. They present an integer linear programming (ILP) formulation to the problem and using techniques such as modulation scaling, a 3-phase polynomial heuristic approach is proposed. The simulation results show us up to 5 times improvement is

achieved by using the ILP based approach and 3.5 times improvement using the 3-phase heuristic approach. Though the results show that the operational time is improved, a relatively narrow formulation is used and this may not work for a more dynamic environment. However, the time cost for solving an integer linear programming problem is prohibitive for large systems [6].

In [7] the authors provide a closed form solution for optimum measurement and reporting time for a single tree network with three different types of data reporting strategies, which are sequential reporting, simultaneous reporting and concurrent measurement and reporting. The results show that the concurrent measurement and reporting strategy provides for smaller finish time as compared to the other strategies due to the fact that some of the nodes may receive almost zero load which effectively reduces the number of active nodes as compared to the concurrent reporting strategy where all the nodes receive some portion of the load. Though the sequential and simultaneous strategies could save energy for few nodes which may not receive any load, this may affect the finish time of a given task. However, the single level architecture (single hop) is not scalable for large scale wireless networks.

**Table 2.1: Cost Calculation for partition of data**

| process | Job | Communication Cost | Time(milliseconds) |
|---|---|---|---|
| p0 | 128*512 | 1 | 65536 |
| p1 | 128*513 | 1.1 | 72089.6 |
| p2 | 128*514 | 1.2 | 78643.2 |
| p3 | 128*515 | 1.3 | 85196.8 |

The experiments described in [6] shows us that, when there is no communication cost or the communication delay is negligible the standard recommended scenario to complete any given task is to allocated the job equally among available processors or nodes. When the communication delay is not negligible then assigning the jobs equally among processors is not optimal. Table 1 shows the calculation to process the data in a 512*512 image array with four identical processes and taking into account 10 percent of the communication delay per pixel and portioning the job equally among the four processes. It also shows that presence of communication delay increases the cost by 30 percent. Also, the experimental results show us that exploiting the arbitrary divisible property of data will improve the performance significantly.

When a load, or a part of it, is communicated to other processors via communication links, the delay (the time it takes to reach the destination) incurred is reflected in the objective function as communication costs. Sensing and communication contribute to the major portion of the energy consumption in a sensor network. Data aggregation aims at eliminating redundancy and minimizing the number of transmissions and thus saves energy. Experiment results from [8], [9], and [10] show that significant energy savings can be achieved with in-network data aggregation.

For any time critical application the data needs to be transferred in a real time. In particular, for a fire detection system or a monitoring system the data has to be delivered to the end user under deadline constraint. The work in [11] provides real time communication architecture by setting priority among packets based on the deadline and by calculating time required by the packet to reach the destination.

The goal of our work is to minimize the overall time for completion of any task, which is reasonable for wireless sensor networks. The reduction of overall time leads to heavy use of sensor nodes in an energy efficient manner. Thus increasing the lifetime of

sensor nodes in a network consequently delivers the required performance. Divisible load theory bears some resemblance to energy based task allocations. However usage of DLT will result in decrease of energy consumption of sensor nodes in a cluster.

## 2.2 Huffman Encoding

The Huffman compression algorithm assumes data that consist of some byte values that occur more frequently than other byte values in the same set of data. By analyzing, the algorithm builds a "Frequency Table" for each byte value within a data set. With the frequency table the algorithm can then build the "Huffman Tree" from the frequency table. The purpose of the tree is to associate each byte value with a bit string of variable length. The more frequently used characters get shorter bit strings, while the less frequent characters get longer bit strings. Thusly the data file may be compressed.

To compress the dataset, the Huffman algorithm reads the data a second time, converting each byte value into the bit string assigned to it by the Huffman Tree and then writing the bit string to a new file. The decompression routine reverses the process by reading in the stored frequency table (presumably stored in the compressed file as a header) that was used in compressing the file. With the frequency table the de-compressor can then re-build the Huffman Tree, and from that, extrapolate all the bit strings stored in the compressed file to their original byte value form.

The Huffman algorithm then builds the Huffman Tree using the frequency table. The tree structure contains nodes, each of which contains a character, its frequency, a pointer to a parent node, and pointers to the left and right child nodes. The tree can contain entries for all 256 possible characters and all 255 possible parent nodes. At first there are no parent nodes. The tree grows by making successive passes through the existing nodes. Each pass searches for two nodes that have not grown a parent node and that have the two lowest frequency counts. When the algorithm finds those two nodes, it

allocates a new node, assigns it as the parent of the two nodes, and gives the new node a frequency count that is the sum of the two child nodes. The next iterations ignore those two child nodes but include the new parent node. The passes continue until only one node with no parent remains. That node will be the root node of the tree.

Compression then involves traversing the tree beginning at the leaf node for the character to be compressed and navigating to the root. This navigation iteratively selects the parent of the current node and sees whether the current node is the "right" or "left" child of the parent, thus determining if the next bit is a one (1) or a zero (0). Because you are proceeding from leaf to root, you are collecting bits in the reverse order in which you will write them to the compressed file. The assignment of the 1 bit to the left branch and the 0 bit to the right is arbitrary.

Decompression involves re-building the Huffman tree from a stored frequency table (again, presumable in the header of the compressed file), and converting its bit streams into characters. You read the file a bit at a time. Beginning at the root node in the Huffman Tree and depending on the value of the bit, you take the right or left branch of the tree and then return to read another bit. When the node you select is a leaf (it has no right and left child nodes) you write its character value to the decompressed file and go back to the root node for the next bit.

In [16] the authors discuss the design issues involved with implementing, adapting and customizing compression algorithm specifically for sensor networks. A large variety of compression algorithms have been proposed including string based techniques [17] [18] [19], entropy based techniques [20] [21] [22] and various others. From [16] it is understood that as the radio energy increases, the benefits of compression also increase significantly. It is also learnt that compression accumulates energy on the compressing nodes and at subsequent nodes from source to sink.

In tinyos1.x the maximum number of bytes which can be sent through a data packet is 128 bytes which includes the 10 bytes header. Tinyos header packed is listed below which can found on the CC2420Radio/AM.h

```
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;
}
```

By default the maximum packed length is also defined in the CC2420Radio/AM.h to 28 bytes which can be modified to accommodate 128 bytes.

```
#define TOSH_DATA_LENGTH 28
```

Sensor networks always have limited amount of energy, therefore to maximize the lifetime of the network and reduce the number of data packets sent across the network it would be appropriate to send more data in a single packet. It is also known that communication cost is higher in terms of energy constraints. Therefore our aim is to increase the payload size of a single packet by using Huffman encoding technique and thus save energy.

# Chapter III Sensing Task Scheduling for Data Fusion

A divisible load is a data parallel load that can be arbitrarily partitioned among links and processors to gain the advantage of parallel processing [13]. Data fusion is generally defined as the use of techniques that combine data from multiple sensors and gather that information in order to achieve inferences, which will be more efficient than if they were achieved by means of a single sensor. Data can be aggregated in-network to reduce communication complexity, and hence energy consumption. The communication cost is several orders of magnitude higher than the computation cost. Directed diffusion can achieve significant energy savings with in-network data aggregation

## *3.1 Sensor Network Architecture*

We consider the sensor network architecture shown in Fig.1. In the architecture sensor nodes are grouped into clusters as intra cluster and inter cluster, and controlled by a single command node: the Base Station (BS). Sensors are only capable of radio-based short-haul communication and are responsible for probing the environment to detect a target/event. Every cluster has a gateway node that manages sensors in the cluster. Clusters can be formed based on many criteria such as communication range, number and type of sensors and geographical location. We assume that nodes are stationary and the Source Nodes (SN) is located within the communication range of its cluster head: the data fusion node (DF).

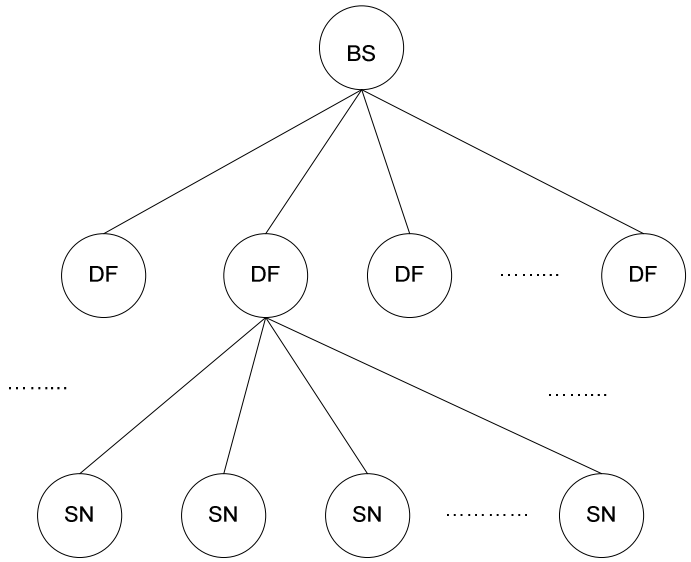


**Figure 3.1 Sensor Network Architecture BS: Base Station; DF: Data Fusion; SN: Sensing Node**

A base station is usually one sensor node connected to the local PC which acts as a simple bridge between the sensor network and the data collection unit. The base station receives the request from the end user and processes the request; it is assumed that the request received from the end-user is converted into a divisible sensing task which is completed by the available nodes in the sensor network. Scheduling of the actual task is done in two stages. First, the base station assigns different workloads to each cluster (i.e. to the data fusion nodes), and then the allocated workload is divided among the available sensors nodes within the particular cluster. The Source nodes measure the data and send it back to the data fusion node, and then the data fusion node, using some aggregation methods such as min, max, average and other methods, depending upon the request from the end-user, sends the fused data back to the base station.

In most sensor nodes the radio hardware uses a single channel, and hence collision occurs when more than one source node send data back to the data fusion node, or when more than one data fusion node sends the fused data back to the base station. On the other hand, when a large task is allocated to the source nodes, there is a possibility

that the data fusion nodes and the base station could be idle for most of the time, while the source nodes report the data back to the data fusion nodes. Therefore, our aim is reduce the idle time of the data fusion node, base station and avoid collisions during communication and minimize the overall time for completion of any task

The sensor nodes are organized into intra and inter clusters. Each intra cluster has exactly one data fusion node responsible for task allocation to source nodes, gathering of data from the source nodes and sending the fused data back to the base station. It is assumed that each intra cluster has an equal number of source nodes. Each source node has a unique identifier and belongs only to one cluster (i.e. one source node transmits data only to one data fusion node). The source node sends the actual or the original data back to the fusion node but, the data fusion node aggregates the data and sends only the required data back to the base station. For example consider a detection scenario; the fusion node only needs to indicate the appearance of a target with a Boolean value. Therefore the fusion node needs only to send the fused data. It is assumed that the links are bi-directional and symmetric. Sensor nodes can perform only one task during a period of time (i.e. a source node cannot transmit data until the assigned measuring is completed), a data fusion node cannot transmit the fused data until all the source nodes have reported their data and processed all the data. It is also assumed that the base station can receive data only from one data fusion node at one point of time, and a data fusion node can receive data only from one of the source nodes in the cluster. Therefore, the communication between all the nodes in the sensor network is always sequential.

Since the communication between the nodes in the network is sequential, it reduces the collision between the nodes and therefore retransmission is not required. Also, the idle time of the data fusion nodes and base station node are reduced. As soon as the data fusion node allocates the sensing task to the source nodes, the source nodes report the data sequentially one by one. This reduces the idle time of the data fusion node

rather than waiting for all the nodes in cluster to complete the task and report the data back to the fusion node, which may result in collisions. The same applies for the base station node where each data fusion node reports data sequentially by reducing collision and idle time rather than waiting for all the fusion nodes to complete the task and report.

Figure 2 illustrates the timing diagram of the intra cluster scheduling, S1, S2…Sn correspond to the sensing nodes from bottom to top and the sensing task (i.e. the workload is allocated in the decreasing order in a cluster from S1 to Sn). The working time of a sensing node can be divided into two parts as the measuring time and the reporting time. Where the measuring time correspond to sensing the actual workload and the reporting time correspond to the sending of the 'actual' data back to the data fusion node. It is assumed that the measuring time and reporting time are constants. The nodes are ordered in such a way that they reflect the actual workload of each sensing node. The node with the highest workload (i.e. the node with the highest sensing task takes a longer time when compared to the other nodes). Thus, a single intra cluster completes the assigned workload when the node with the highest workload completes its task.

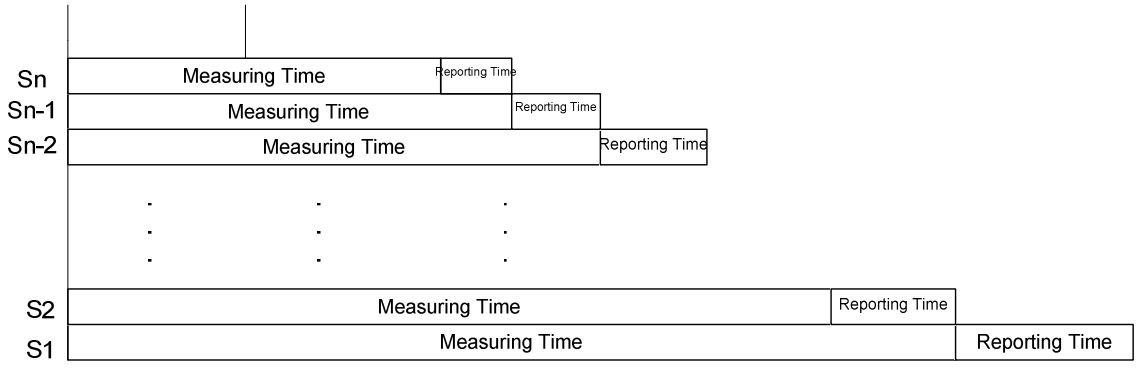


**Figure 3.2: Timing Diagram Intra-Cluster [12]**

In general for an ith cluster in the network the following recursive equations are obtained [12]

$$m_{i,1}S_m = m_{i,2}S_m + m_{i,2}S_c \quad (1)$$

$$m_{i,2}S_m = m_{i,3}S_m + m_{i,3}S_c$$

$$m_{i,n-1}S_m = m_{i,n}S_m + m_{i,n}S_c$$

$$m_{i,j} = H^{(j-1)} m_{i,1}, \; j=2\ldots..n, \quad (2)$$

Where

$$H = \frac{S_m}{S_m + S_c}$$

$$M_i = m_{i,1} + m_{i,2} + \ldots.. + m_{i,n} = \sum_{j=1}^{n} m_{i,j} \quad (3)$$

From the above equations the node with the largest workload can be assigned based on this equation.

$$m_{i,1} = \frac{M_i}{1 + \sum_{k=1}^{n-1} H^k} \quad (4)$$

Similarly the workloads for other nodes can be assigned based on the following equation.

$$m_{i,j} = H^{(j-1)} \frac{M_i}{1 + \sum_{k=1}^{n-1} H^k} \quad (5)$$

As stated earlier, the time taken by the node with largest workload is the time required for the cluster to complete the assigned task.

$$t_i = \frac{M_i}{1 + \sum_{k=1}^{n-1} H^k} (S_m + S_c) \quad (6)$$

Figure 3 illustrates the timing diagram of the inter cluster scheduling, where $DF_1$, $DF_2$….. $DF_N$ corresponds to the data fusion node. The working time of a data fusion node can be divided into two parts as the processing time and the reporting time. Where the processing time correspond to producing a fused output data through some form of transformation based on the requirements and the reporting time correspond to the fused data to the base station. Each fusion node starts processing the data from the sensing node at time $t_{i,}$ which depends on the workload assigned to each intra cluster. As soon as the data fusion node receives the data from the source nodes, the data fusion nodes report the data sequentially to the base station. Thus reducing the idle time of the base station and hence avoids collisions.

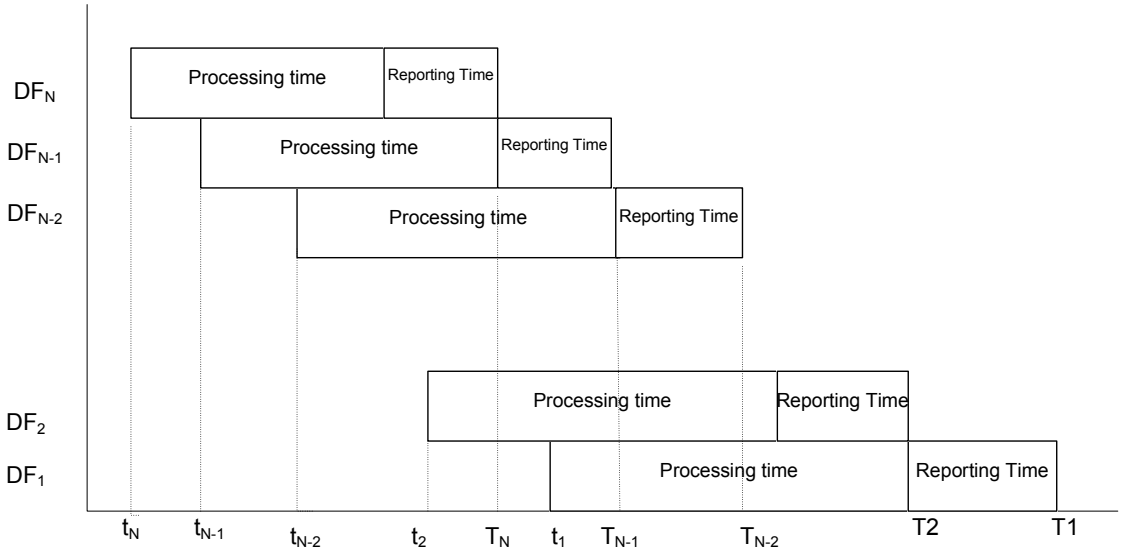


**Figure 3.3: Timing Diagram Inter-Cluster [12]**

$T_1, T_{2,}.....T_N$ corresponds to the finish time of each data fusion node, thus the minimum finish time of the whole network can be given as

$$T = Max(T_1, T_{2,}.....T_N) \tag{7}$$

From equations (4) we can get

$$t_1 + M_1 S_p = t_2 + M_2 S_p + f(M_2) S_c \tag{8}$$

$$t_2 + M_2 S_p = t_3 + M_3 S_p + f(M_3) S_c$$

Generally we can say

$$t_{N-1} + M_{N-1} S_p = t_N + M_N S_p + f(M_N) S_c$$

Using (6) $t_i (i = 1....N)$ is expressed as

$$t_i = C M_i \tag{9}$$

Where $$C = \frac{(S_m + S_c)}{1 + \sum_{k=1}^{n-1} H^k}$$

Workload assigned to each cluster can be given by

$$M_1 = \frac{f(M_2) S_c}{C + S_p} + M_2 \tag{10}$$

$$M_2 = \frac{f(M_3) S_c}{C + S_p} + M_3$$

$$M_{N-1} = \frac{f(M_N) S_c}{C + S_p} + M_N$$

$$M_i = M_1 - \frac{S_c}{C + S_P} \sum_{j=1}^{i-1} f(M_j), i = 2.....N \tag{11}$$

M corresponds to the total workload assigned required to complete a given task.

Equation (11) corresponds to the solution of each task when the aggregation function is known.

$$M = \sum_{i=1}^{N} M_i \tag{12}$$

Substituting (11) in (12) we get

$$M = N M_1 - \frac{S_c}{C + S_P} \sum_{i=2}^{N} \sum_{j=1}^{i-1} f(M_j) \tag{13}$$

$$M_1 = \frac{1}{N}\left( M + \frac{S_c}{C + S_P} \sum_{i=2}^{N} \sum_{j=1}^{i-1} f(M_j) \right) \qquad (14)$$

Since the above strategy obtains a minimum finish time at the intra-cluster as well as inter-cluster level, the overall finish time of the task is also minimized.

An implementation of choosing the node to assign with the maximum payload within the cluster and among clusters is listed below.

BS sends a Broadcast message to all nodes requesting for voltage level of each nodes.The SN's send the voltage level to the DF nodes which in turn sends a value based on combined values of its own and value of the nodes in the cluster to the BS node. BS on receiving the voltage value using the inter-cluster algorithm assigns the maximum payload task to the DF node with the highest voltage. The intra-cluster algorithm assigns the maximum payload task to sensing nodes based on the voltage values already acquired.

Anytime a new task is assigned to the whole system, the BS node sends a broadcast message to calculate the power levels on all nodes and in each cluster. So that, the BS node can assign the task using DLT based on the power levels in the cluster. Though the time for communication and processing for these three messages is very negligible this overhead is ignored by assuming this is a concurrent system thread for monitoring each node's energy residue.

Once the base station node has received the cluster energy information, the BS uses the DLT algorithm to assign the workload to each cluster based on the energy in each cluster.

BS along with the workload information it also sends a result bit along with the workload information. The result bit informs the data fusion node how the base station requires the result of the workload. The result could either be a single value i.e. it could be as simple yes or no for an intrusion detection system or it could be something like a maximum temperature value of that cluster or it could request the whole data i.e. the original requested data for example 100 sample readings of temperature in a particular area. The base station on other hand could use the original data to store the data elsewhere and then use the data for future analysis.

Once the Data fusion node receives the request from the base station, using the DLT intra cluster algorithm the data fusion node allocates the task to the source nodes. Once again the highest task is assigned to the node with the highest energy. This information was stored when the base station sent a request for the energy levels in each cluster. Each source node sends its energy levels to the data fusion node, it stores this information and based on the energy levels the task is assigned to the source nodes. Unlike the base station, the data fusion node does not send a request bit. It directly requests for the number of readings required from the source node.

## *3.2 Analysis of Compression on sensor nodes*

**Table 3.1: Effect on frequency of data for compression**

| | No of Encoded bytes | | |
|---|---|---|---|
| **No of actual bytes of data** | **High frequency** | **Moderate freq** | **Low Freq** |
| 5 | 10 | 14 | 34 |
| 25 | 13 | 40 | 49 |
| 50 | 16 | 43 | >118 |
| 100 | 22 | 50 | >118 |
| 200 | 34 | 102 | >118 |
| 400 | 59 | >118 | >118 |
| 700 | 97 | >118 | >118 |
| 800 | 109 | >118 | >118 |

Table 2 indicates the compression ratio using Huffman encoding technique on the sensor nodes. A maximum of 118 bytes of data can be transmitted from one node to another node. By using Huffman encoding and depending on the frequency of data, more bytes can be sent in a single packet. From the above table it is learnt that on a best case scenario i.e. higher the frequency of the data around 800 bytes of actual data can be transmitted from one node to another node. Without compression it would requires 7 packets to send 800 bytes of data from one node to another node. This saves a lot of energy. It is learnt from several experiments that computational cost is always less than communication cost. Though the time taken to encode the actual data and then decode the original data is approximately 2-4 seconds, this saves a lot of energy.

Even on a moderate frequency rate we should be able to send 200 bytes of data in a single packet. As the number of bytes to encode and decode decrease the overall time for encoding and decoding becomes almost negligible and still be able to send twice the number of packets. We improve the performance significantly by saving energy and improving the lifetime of the whole network.

Another issue to be considered here is, only a maximum of 118 bytes can be transferred in one packet. So, using the compression it is possible to send lesser number of encode bytes. For example let us consider the actual number of bytes is 100, and the encoded length is 20 bytes. The total number of meaningful bytes sent in a packet will be 30 and the remaining 98 bytes of data will be padded by zero's which is same as sending the actual 118 bytes of data. To avoid this, the encoding string length is calculated and till the encoded string length reaches 118 bytes a packet will not be sent. This helps avoid unnecessary usage of radio, padding of the remaining bytes, and thus saving energy.

From Table 3.1 it is learnt that it is also possible to send more encoded bytes than the original data bytes. This could occur if the frequency in the data is minimum or if the

actual data packet size is too low. To avoid this issue, after the encoding is completed the actual data size and the encoded data length is compared and if the encoded data size is greater than the actual data size the original data packet is sent directly without encoding. Here, it is possible to argue that encoding the actual data and comparing them with encoded length could be unnecessary waste of computation time and energy but we understand that computation time is almost negligible and will not result in a setback. To differentiate between the actual message the encoded message a bit is set in the packet to identify if the received packed is encoded or the actual message packet.

The source nodes receives the request from the data fusion node, depending upon the application, the source node completes the sensing task and encodes the data and compares with the actual data size. If the encoded data length is less than the actual data and is close to 118 bytes of data that can be sent in a single packet the data will be sent with encoded bit set. If the encoded data length is greater than actual data length, the actual data close to 118 bytes will be sent with encoded bit set to 0.

Once the data fusion node receives the data from all the source nodes within the cluster, it processes the data. Processing could be any mathematical operation completed depending on the requirement of application (request sent by the base station). It could be a single byte data or the whole data. If the request from the base station requested the original data then the data fusion node could still again use the compression technique to send data to the base station.

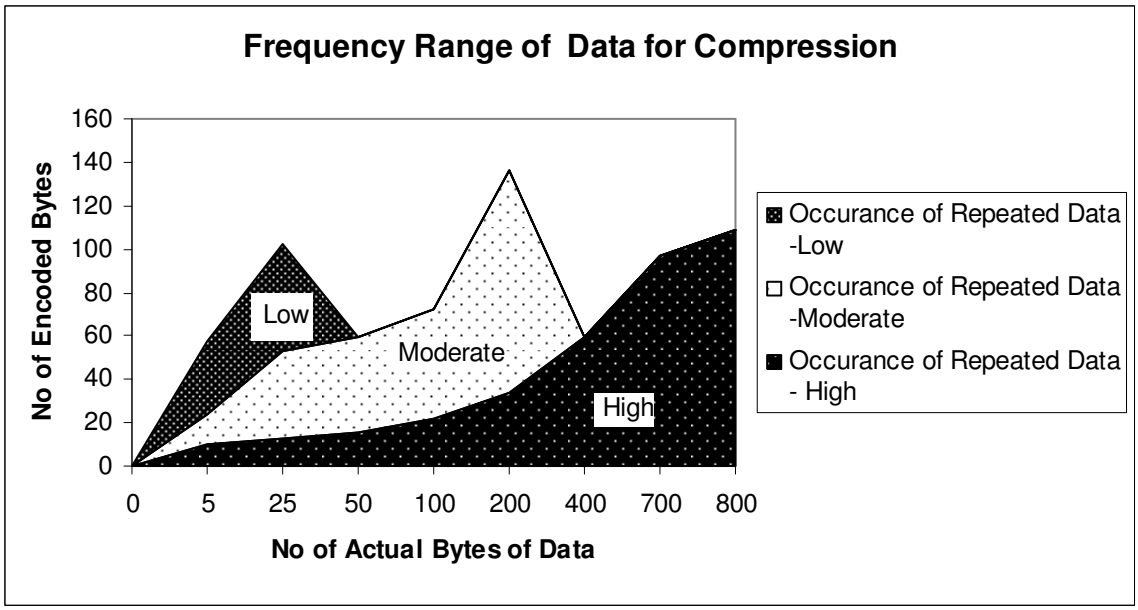


**Figure 3.4: Frequency Range of Data for compression**

Data can be grouped into three categories such as data with high, moderate and low frequency. If the frequency of data is low then it is not appropriate to use compression on the data as the encoded data size is large than the actual data size. It is clearly evident from Table 3.1 that if the frequency is too high then compression could be used to transmit the data across the network. Therefore compression techniques can be used for most sensor networks with moderate or high frequency of data. By using compression techniques it possible to significantly increase the performance of the network and thus increases the lifetime of the system by saving energy.

## 3.2 Analysis and Experimental Results

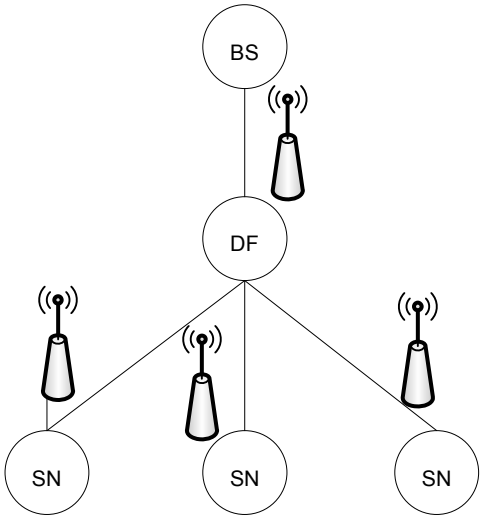


**Figure 3.5: Experimental Setup BS: Base Station; DF: Data Fusion; SN: Sensing Node**

The experimental setup fig 3.4 consists of the three sensing nodes, one data fusion node and one base station. The experiments were conducted for a single intra-cluster which consists of three sensing nodes and one data fusion node and the aggregation function is hard coded as the requirements of the user. Experiments were conducted on the tmote sensor nodes using tinyos 1.0 version.

**Table 3.2: Readings with Varying Payloads**

| No of Readings | Total Bytes/Data per packet | Sensing and Reporting | Sensing | Reporting |
|---|---|---|---|---|
| 100 | 15/5 | 26059 | 259.32 | 1.27 |
| | 25/10 | 26003 | 259.31 | 0.72 |
| | 45/20 | 26048 | 260.48 | 0.44 |
| | 55/25 | 26043 | 260.48 | 0.39 |
| | 105/50 | 25935 | 259.07 | 0.3 |
| | | | | |
| 200 | 15/5 | 50068 | 248.04 | 1.3 |
| | 25/10 | 50143 | 250.25 | 0.69 |
| | 45/20 | 50087 | 250.25 | 0.41 |
| | 55/25 | 49877 | 249.18 | 0.255 |
| | 105/50 | 50057 | 250.25 | 0.26 |
| | | | | |
| 300 | 15/5 | 74011 | 245.38 | 1.32 |
| | 25/10 | 73915 | 245.68 | 0.7 |
| | 45/20 | 73829 | 245.68 | 0.41 |
| | 55/25 | 74078 | 246.67 | 0.25 |
| | 105/50 | 73774 | 245.68 | 0.23 |
| | | Average | 251.7147 | 0.596333 |

For a given task to be completed in the minimum finish time based on the strategy presented, the sensing time and the reporting has to be calculated to assign to the intra-cluster nodes i.e. the sensing nodes. Therefore to calculate the sensing and reporting time, various experiments were conducted. Table 1 shows various experiments done to calculate the sensing time and reporting time/communication time by varying few

parameters. The parameters which were considered are the number of readings required by the data fusion node, total bytes/the number readings to be sent in a single packet. The maximum payload/data that can be sent by a sensor node (CC2420) is 118 bytes.

**Table3.3 : Readings with Varying Bytes/Message**

| | No of Readings | | | Average (Milliseconds) |
|---|---|---|---|---|
| Total bytes/ Message | 100 | 200 | 300 | |
| 15 | 1.27 | 1.3 | 1.32 | 1.296667 |
| 25 | 0.72 | 0.69 | 0.7 | 0.703333 |
| 45 | 0.44 | 0.41 | 0.41 | 0.42 |
| 55 | 0.39 | 0.255 | 0.25 | 0.298333 |
| 105 | 0.3 | 0.26 | 0.23 | 0.263333 |

Table 3.3 shows the effect of communication time $S_c$ by increasing the payload size with regards to number of bytes/message. Communication time $S_c$ against the finish time is plotted in figure 3.6. It is clearly understood that as the number of bytes/message increases the time required to send the data decreases i.e. as the size of the payload increases, the time (communication time) taken to send the message decreases. The results show that as the communication speed increases the finish time of the task can also be reduced. Hence it is necessary to send the maximum payload in a single packet which results in effective usage of the bandwidth.

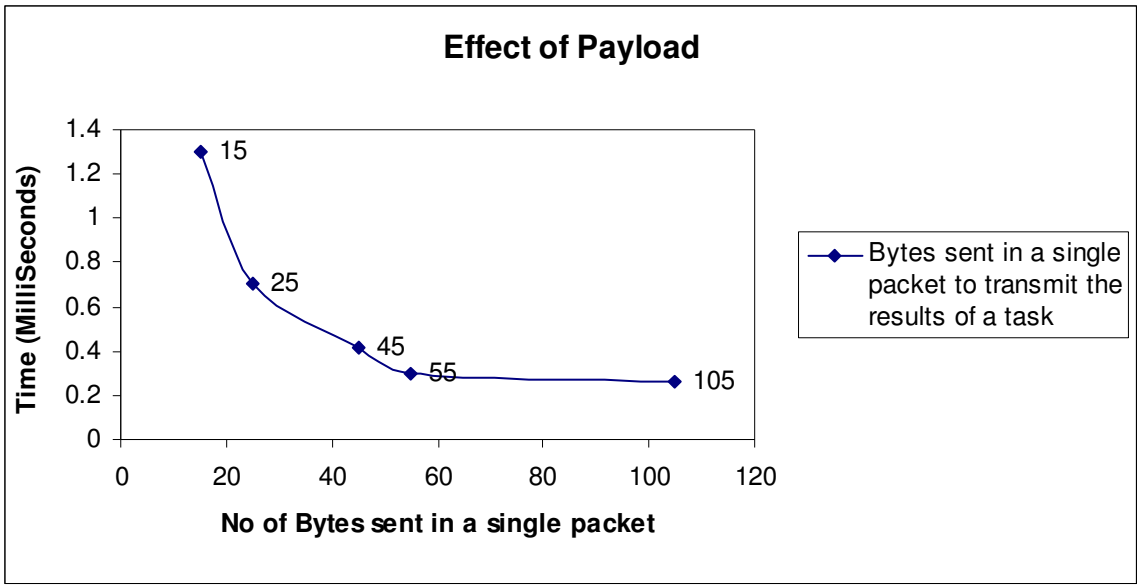


**Figure 3.6: Effect of Payload**

Table3.2 show the average time taken to sense the required data. Experiments were conducted by varying the number of readings to be sampled. From Table 3.4 it is evident that as the number of readings required by the sensing node increases the time also increases which shows that the number of readings required and time are linear. Hence we justify that the sensing node with the maximum number of readings take a longer time than the other nodes with fewer readings and thus making sure the nodes follow the sequential computation and communication model.

**Table 3.4: Measuring Time/Sensing time $S_m$**

| No of Readings | Time (Milliseconds) | Time(Milliseconds) / Reading |
|---|---|---|
| 50 | 14105 | 282 |
| 100 | 27105 | 271 |
| 150 | 38230 | 254 |
| 200 | 50205 | 251 |
| 300 | 74243 | 247 |
| 500 | 125005 | 250 |
| 1000 | 247006 | 247 |
| **Average** | | **257.42** |

Table 3.5 shows the average reporting time required to transmit the required data to the data fusion or the base station. The reporting time cannot be calculated separately; therefore the sensing time and the reporting time were calculated together and then reporting time were calculated based on the results of Table 3.4

**Table 3.5: Reporting Time/Communication Time Sc**

| No of Readings | No of Readings /Message | Sensing + Reporting time | Reporting Time (Milli Seconds) |
|---|---|---|---|
| 50 | 5 | 14397 | 288 |
| 100 | 5 | 26548 | 265 |
| 150 | 5 | 38725 | 258 |
| 200 | 5 | 50888 | 254 |
| 300 | 5 | 75208 | 251 |
| 500 | 5 | 123848 | 248 |
| 1000 | 5 | 245448 | 245 |
| Average | | | 258.42 |

Calculation: Task allocation of sensing nodes within an intra-cluster:
From Table 3.4: Average Measuring time $S_m$ = 257 Milliseconds
From Table 3.5: Sensing time + Reporting Time = 257.6MilliSeconds
Reporting Time $S_c$ = .001MilliSeconds
The total Workload (M) assigned for a single intra-cluster is 100.

Since there are 3 sensing nodes, the sensing task has to be allocated between the 3 nodes. Allocating a node with the largest workload within a cluster can be based on the node id. Choosing a node to allocate maximum workload in a cluster can be done based on energy levels. The node with the highest energy level can be chose. Before assigning the task to a sensing node, the data fusion node requests the energy levels at all the nodes in a cluster and the node with the highest energy level will be chose to allocate the maximum load. The remaining nodes, on the energy levels will be allocated the required task. Thus all the nodes in the cluster will be used equally rather than overloading a single node and increasing the lifetime of the cluster.

Using (4) for the node with largest workload

$m_{i,1}$ = 33.6 ~34 readings

Using (5) for the remaining nodes the workload assigned are

$m_{i,2}$ = 32.9 ~33readings

$m_{i,3}$ =32.2 ~32readings

$T_i$ = 8.66 Seconds

Where i refer to the intra-cluster and 1, 2, 3 refers to the nodes within in a particular cluster. The time taken to complete the total task is 8.66 seconds which is far better than the time required to sense 100 readings i.e. 26.5 seconds.

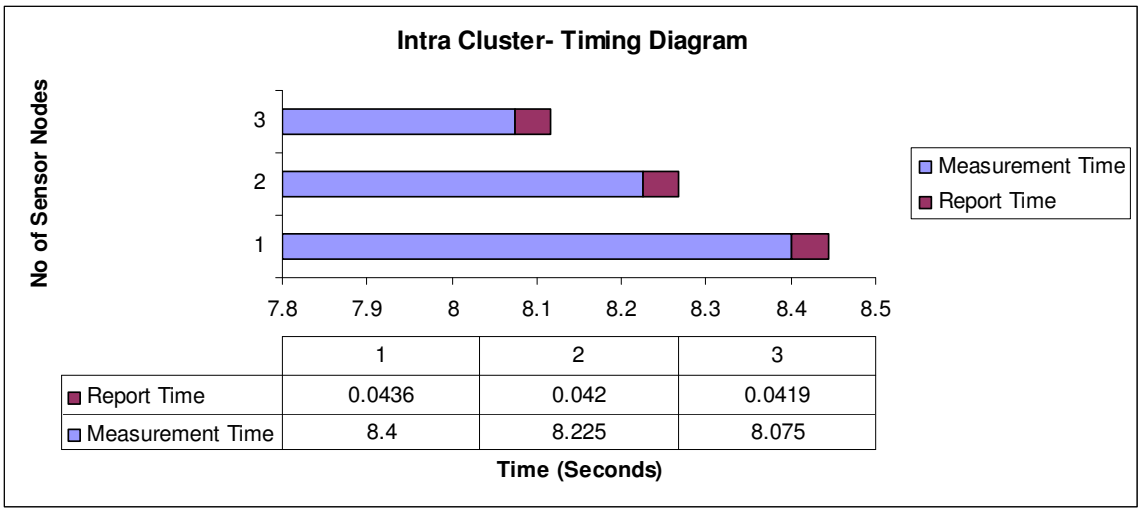


**Figure 3.7: Timing Diagram intra-cluster**

From figure 3.6 it is clearly evident that the completion of task depends on allocation of the task to a particular node. When the node with the largest workload completes its sensing task, the intra-cluster has completed the sensing task allocated by the data fusion node. It is visible that the proposed algorithm follows the sequential computation and communication model. The non-overlapping regions of the reporting show us that there are no collisions. The gap between completion of task between one node and another node is the order of milliseconds which is almost negligible and quite acceptable. This reduces the idle time of the data fusion node. Since idle wait time of the nodes are reduced completely and there are no collisions, a minimum finish time for any given task can be obtained.

# Chapter IV Application of Sensor Network

"Precision agriculture concentrates on providing the means for observing, assessing and controlling agricultural practices". Recent developments in the wireless sensor fields have led to the emergence of Environmental Sensor Networks (ESN) .ESN greatly enhance the monitoring of the natural environment and in some cases open up new techniques for taking measurements or allow previously impossible deployments of sensors [14]. Environmental sensor networks aids in collection, analysis of data, and allow the researcher to observe the environment from their office. ESN represent a new way to sense and understand the environment, which have a huge potential in many areas of the environmental sciences. ESN have been used for various environmental monitoring applications such as weather monitoring, fire detection, etc. Wireless sensor networks can be applied effectively for monitoring of the agriculture field. Application specific sensors and sensor nodes need to be tailored specifically to the environment being sensed.

## *4.1 Problem Description*

Data loggers have been used in the agriculture field to collect the soil moisture data at regular intervals which require manual downloading of data by maintaining a team. Our application concentrates on providing soil moisture data to the researchers eventually helping them to take necessary action in the agriculture filed based on the recorded data. The main goal of the application is to monitor the agriculture field for soil moisture data helping farmers to provide the required treatment to field or parts of the agriculture field based on the collected data and to collect data automatically.

## *4.2Our Approach*

The monitoring of the agriculture field can be effectively done with wireless sensor networks. Various reasons for adapting to Wireless sensor networks include

reduction in wiring, faster deployment and installation, mobility, robustness, low-cost, small size, low power management and requires less human intervention [15].

## 4.3Hardware and Software Requirements

- Sensor nodes (tmote from Moteiv Corporation)
  - 2.4GHz chipcon Wireless Transceiver
  - MSP430 Microcontroller (Texas instruments)
  - 1MB external Flash
  - Integrated ADC, DAC, Supply Voltage Supervisor, and DMA Controller
  - Integrated humidity, temperature and light sensors
- Soil Moisture sensors
  - Measurement time: 10ms
  - Accuracy: With soil specific calibration +/- 2%
  - Power Requirements: 3VDC @ 10mA
  - Output excitation voltage: 2500mV excitation
  - Operating environment: $0 - 50^0 C$
- Tinyos operating system: nesC programming language

## 4.4 Challenges for Precision Agriculture

- Tmote sensor nodes have been used for data collection. Tmote sensor nodes have humidity, temperature and light sensors integrated with the node. Sensor nodes do not include soil moisture sensors. Therefore soil moisture sensors have to be integrated to the tmote (done by my colleague Zhen Li).
- Tmote consists of 2 DAC's and 4 ADC's, to minimize the use of number of sensor nodes each sensor node has to be integrated with 4 soil moisture sensors (done by my colleague Zhen Li).
- Efficient storage and retrieval of data from external flash

- Integrating soil moisture sensors to tmote physically as well as programmatically.
- Power Management i.e. to test energy consumption and replacing the batteries
- Packing the hardware as an entire unit (done by my colleague Zhen Li).

## *4.5 Experimental setup*

Figure 4.1 represents the experimental setup of the precision agriculture project. The setup consists of 10 sensing nodes with each sensor node connected with 4 soil moisture sensors, 1 central node and one base station node connected with the laptop to store the data. Each sensing node samples the soil moisture sensors (4) at a regular interval of 2 hours. Sensing node stores the data locally in the ram each day and at the end of each day the sensing nodes send the soil moisture readings (4) along with the current voltage readings. The voltage readings are recorded because the sensor nodes battery has to be replaced for efficient working of soil moisture sensors.

The data from the sensing nodes are stored on central nodes flash. Tmote consists of 1MB of external flash. Flash is divided in to 16 blocks, each block consisting of 256 pages and each consisting of 256 bytes. Each sensing node sends 100 bytes of data per day. We can use each page to store each sensing nodes data per day. Therefore for storing 10 nodes data, external flash can accommodate to store at least for a period of 21 days provided the battery still holds good. Based on the sensor node's ID each page is allocated to store the sensing nodes data i.e. for node id 1, pages from 0-7 can used to store a weeks data and for node id 2, pages from 8-14 can be used. The central node retrieves the node id from the incoming message and based on the node id the data is stored in the flash. Using the base station the data from the central node can be retrieved efficiently and stored for processing by the researchers.

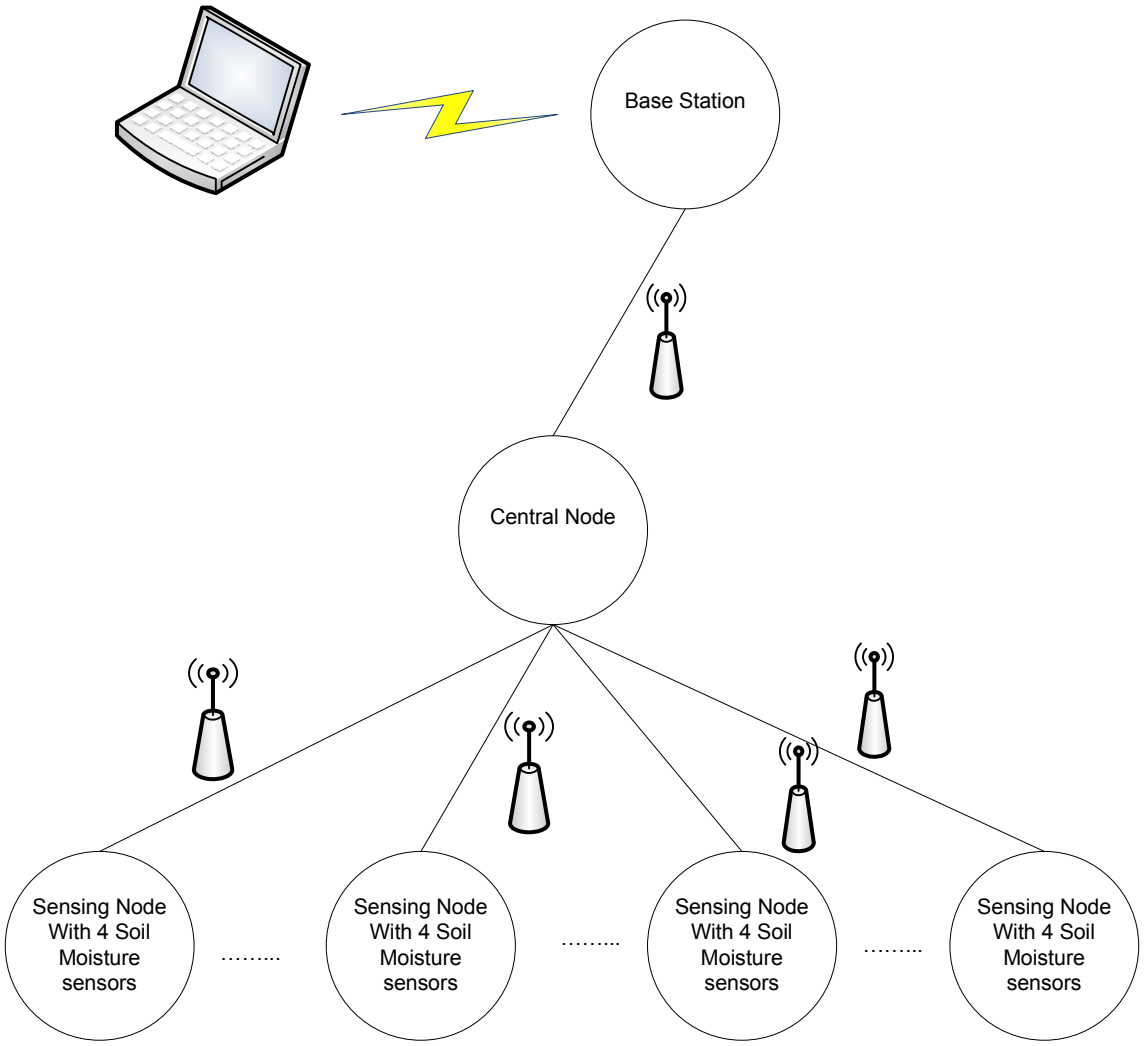


**Figure 4.1: Experimental Setup Precision Agriculture**

Radio on the sensor nodes consumes a lot of energy comparatively. The sensing nodes are required to transmit data to the central node only once a day and do not require to listen for any messages from other nodes. Therefore, shutting down the radio on the sensing nodes other than the time to send one message per day will increase the battery life of the sensor network.

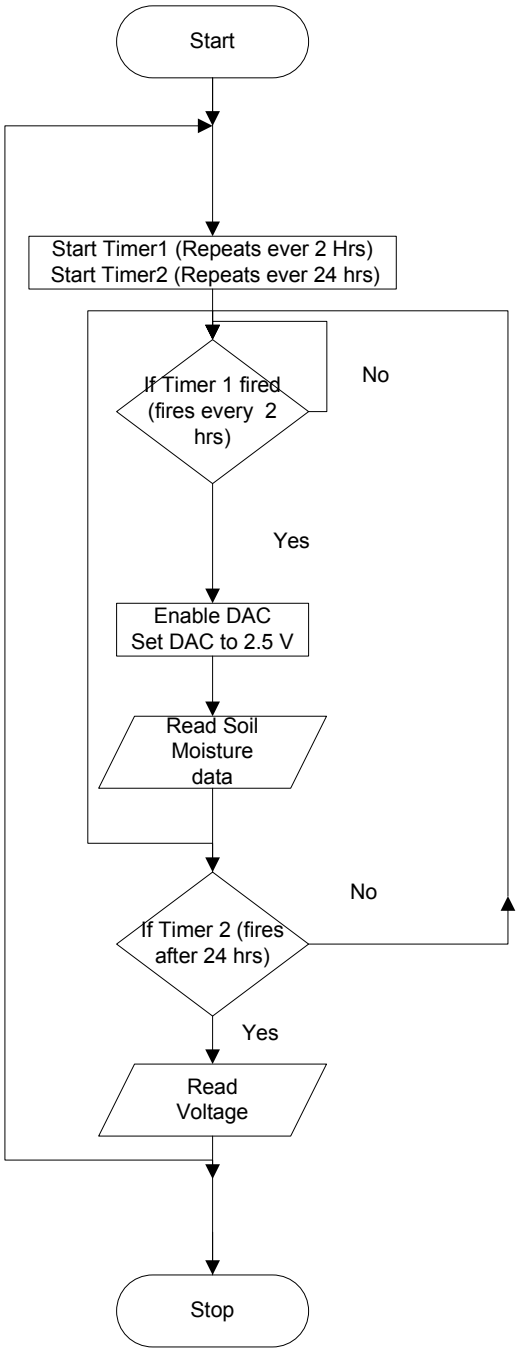


**Figure 4.2: Flowchart Sensing Node**

Total bytes = (12 readings*2 bytes * 4 sensors) +10 bytes header+2 bytes node id +2 bytes voltage

=96+10+2+2

=110 bytes/day/node

Maximum bytes that can be sent in 1 message is 128 bytes

Therefore the Packet structure (single packet) is as follows

uint16_t adc_zero[6]; 2bytes*12=24 bytes

uint16_t adc_one[6]; 2bytes*12=24 bytes

uint16_t adc_two[6]; 2bytes*12=24 bytes

uint16_t adc_three[6]; 2bytes*12=24 bytes

Uint16_t volt; 2 bytes

uint16_t node_id; 2 bytes

---

Total = 100 bytes+10 bytes header=110 bytes

The setup of this project looks similar to the 2 level DLT with a single cluster which could use the intra cluster scheduling algorithm to complete the task in a shortest time. From the experiments conducted for soil moisture readings it is seen that the frequency of data is high. Data is transmitted from the sensing nodes every day once over a period of several days; as we already know communication cost decreases the lifetime of the network due to the limited amount of energy on the deployed nodes, and as the application requires large amount of data, it would be appropriate for us to use compression technique already discussed to reduce the number of messages sent across the network to transfer the data. By using the compression technique discussed early increase the lifetime of the network by using the radio of the sensor nodes less frequently which eventually leads to energy savings.

# Chapter V Conclusion & Future Work

This chapter summarizes the thesis and the future work that can be done further is discussed.

## *5.1 Conclusion*

The proposed novel algorithms schedule a set of computational tasks, which may have dependencies and communication, into a set of heterogeneous processors in such a way that minimizes both the total consumed energy and the make life span of completion of the task shorter (i.e., the time required having all tasks completed). Radio on the sensor nodes consumes a lot of energy comparatively. For energy critical applications involving in large amount of data it would helpful to reduce the number of messages transmitted or by increasing the number of data that can be sent in a single packet across the network.

Experiments show that significant improvements can be achieved by using the proposed algorithms to complete any given task in minimum finish time. Its is also learnt that using compression techniques more than 118 bytes of data can be transferred from one node to another node depending upon the application. Even though lot of computation energy is required to encode and decode the actual data, computational cost is far less then the communication cost on the sensor nodes. Experiments shows us that on an average at least twice the actual data can be sent in a single packet which improves the performance significantly by saving energy and increasing the lifetime of the actual network.

## *5.2 Contribution to Knowledge*

Most of the compression techniques proposed in sensor networks so far have been application specific. Several techniques such as string matching, entropy based encoding techniques have been proposed and implemented. This thesis addresses the power

management issues by applying Huffman encoding techniques on the sensor data to improve the number of bytes sent in a single packet and thus improving the global power consumption of the network

## *5.3 Future work*

Future extensions to this work can consider,

- Large scale deployment of sensor nodes with compression for processing large amounts of data
- Using 2 level DLT in Body sensor networks applications to complete the task in the shortest time

# Reference


1. Koutsakis, P.; Papadakis, H. "*Efficient medium access control for wireless sensor networks,"* Wireless Pervasive Computing, 2006 1st International Symposium on, Vol., Iss, 16-18 Jan. 2006 Page: 5

2. B. Veeravalli, D. Ghose, V. Mani, and T. G. Robertazzi, "*Scheduling Divisible Loads in Parallel and Distributed Systems*." Los Almitos, California: IEEE Computer Society Press, 1996

3. Robertazzi, T.G. "*Ten reasons to use divisible load theory"* Computer, Vol.36, Iss.5, May 2003 Pages: 63- 68

4. H. O. Sanli, R. Poornachandran, and H. Cam, "*Collaborative two-level task scheduling for wireless sensor nodes with multiple sensing units*," in *IEEE SECON 2005 proceedings*, 2005, Pages: 350–361

5. Yu, Y. and Prasanna, V. K. 2005. *"Energy-balanced task allocation for collaborative processing in wireless sensor networks". Mob. Netw. Appl.* 10, 1-2 (Feb. 2005), 115-131.

6. Yang Yu; Bhaskar Krishnamachari; Prasanna, V.K. "*Issues in designing middleware for wireless sensor networks"* Network, IEEE, Vol.18, Iss.1, Jan/Feb 2004 Pages: 15- 21

7. Moges, M.; Robertazzi, T.G. "*Wireless sensor networks: scheduling for measurement and data reporting"* Aerospace and Electronic Systems, IEEE Transactions on, Vol.42, Iss.1, Jan. 2006 Pages: 327- 340

8. Intanagonwiwat, C.; Estrin, D.; Govindan, R.; Heidemann, J.*; "Impact of network density on data aggregation in wireless sensor networks"* Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on
2-5 July 2002 Pages: 457 – 458

9. Seung Jun Baek; Gustavo de Veciana; Xun Su "*Minimizing energy consumption in large-scale sensor networks through distributed data compression and hierarchical aggregation"* Selected Areas in Communications, IEEE Journal on, Vol.22, Iss.6, Aug.

2004 Pages: 1130- 1140

10. Krishnamachari, L.; Estrin, D.; Wicker, S. "*The impact of data aggregation in wireless sensor networks*" Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on, Vol., Iss., 2002 Pages: 575- 578

11. C. Lu, B. Blum, T. F. Abdelzaher, J. Stankovic and T. He,"*RAP: A Real-Time Communication Architecture for Large-Scale Wireless Sensor Networks*", *Proc. of Eight IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002

12. H. Kang and X. Li, "*Sensing Workload Scheduling in Hierarchical Sensor Networks for Data Fusion Applications"* ACM International Wireless Communications and Mobile Computing Conference (IWCMC), Honolulu, Hawaii, 2007.

13. T.G. Robertazzi and D. Yu," *Multi-Source Grid Scheduling for Divisible Loads*," 2006 Conference on Information Sciences and Systems, Princeton University, Princeton N.J, March 2006

14. Martinez, K.; Hart, J.K.; Ong, R. "*Environmental sensor networks: A revolution in the earth system science*" Computer Volume 37, Issue 8, Aug. 2004 Pages: 50 – 56 Digital Object Identifier 10.1109/MC.2004.91

15. Ning Wang, Naiqian Zhang and Maohua Wang. "*Wireless sensors in agriculture and food industry— Recent development and future perspective"* Computers and Electronics in Agriculture, Volume 50, Issue 1, January 2006, Pages: 1-14

16. C. Sadler and M. Martonosi, "*Data compression algorithms for energy-constrained devices in delay tolerant networks,*" in *Proc.* ACM Conf. on Embedded Networked Sensor Systems, 2006.

17. Markus Oberhumer. LZO Real-Time Data Compression Library. http://www. oberhumer.com/opensource/lzo/,Oct. 2005.

18. J. A. Storer and T. G. Szymanski.*" Data Compression via Textual Substitution."* Journal of the ACM, 29:928–951, 1982.

19. J. Ziv and A. Lempel. *"A Universal Algorithm for Sequential Data Compression."* IEEE Transactions on Information Theory, 23(3):337–343, 1977.

20. J. G. Cleary and I. H.Witten. *"Data Compression using Adaptive Coding and Partial String Matching." IEEE Transactions on Communications, COM-32(4):396–*

*402, April 1984*

21. D. A. Huffman. *"AMethod for the Construction ofMinimum-Redundancy Codes."* In Proceedings of the I. R. E., 1952

22. I. Witten, R. Neal, and J. Cleary. *"Arithmetic Coding for DataCompression"*. Commun. ACM, 30:520–540, June 1987

# Appendix A DLT Code

***List of Files***

***MyBaseStation node***

- *MyBaseStationM.nc*
- *MyBaseStation.nc*
- *Bs_Df_InterClusterMsg.h*
- *Df_Bs_ReceiveIntraVoltageMsg.h*
- *MyBrMsg.h*
- *Makefile*

***Data Fusion node***

- *DatafusionM.nc*
- *Datafusion.nc*
- *Df_Bs_IntraVoltageMsg.h*
- *Bs_Df_ReciveClusterTaskMsg.h*
- *Df_Sn_IntraClusterMsg.h*
- *Sn_DF_ReceiveVoltageMsg.h*
- *Makefile*

***Source node***

- *SourceNodeM.nc*
- *SouceNode.nc*
- *bs_sn_broadcas.h*
- *Df_Sn_ReciveClusterTaskMsg.h*
- *DiffuseMsg.h*
- *Sn_Df_powerMsg.h*
- *SourceMsg.h*
- *Sn_Df_dataMsg.h*
- *Makefile*

*\cygwin\opt\moteiv\apps\TOSBASE application has been used to receive all the broadcast messages and for testing purposes.*

Components diagram for all the modules have been created using the nesdoc tool available in tinyos. The command used is "*make tmote nesdoc*" under the application directory.

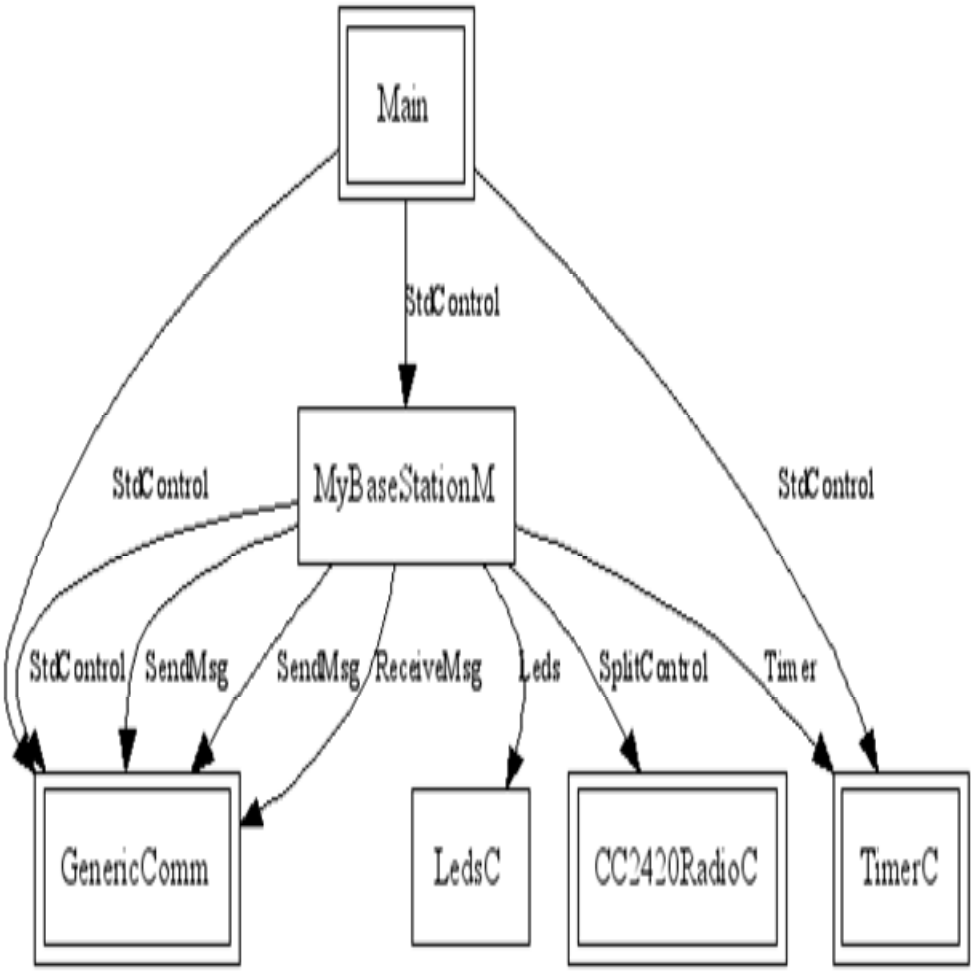


**Figure 5: MyBaseStationComponents Wring**

***MyBaseStationM.nc***

```
/****
 This program acts as the Base Station Node which sends a message
 intially for get energy values, then assign the task to the clusters based on the scheduling
 algiorithm
**/
includes MyBrMsg; //message to get energy values of the clusters
includes Df_Bs_ReceiveIntraVoltageMsg; //message used for BS node to receive the
voltage values
includes Bs_Df_InterClusterMsg; //message to get the result from the data fusion node
```

***MyBaseStationM.nc***

```
module MyBaseStationM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer; //to calulate the time
        interface StdControl as CommControl; // for sending and receive messages
        interface SendMsg as Send;
        interface SendMsg as Send1; //bs_df_intercluster
        interface ReceiveMsg as Receive;
        interface Leds;// for leds
        interface SplitControl as RadioControl;

  }
}
implementation {
        TOS_Msg pkt; // for sending broadcast messages
        TOS_Msg pkt1; //for sending task allocation messages
        uint16_t i;
        //for calculation of voltage of all Datafusion nodes
        uint16_t volt_df1;
        uint16_t volt_df2;
        uint16_t volt_df3;
        //for calculation inter cluster values and setting priority
        uint16_t inter_p1;
        uint16_t inter_p2;
        uint16_t inter_p3;
        //To assign nodes as datafusion
        uint16_t df_node1;
        uint16_t df_node2;
        uint16_t df_node3;
        uint16_t total_nodes=3;
        uint16_t total_workload=1000;
```

**MyBaseStationM.nc**

```
        float h=0.99;
        float temph=0.99;
        float deno=0;
        //to check if task has been allocated
        bool flag_df1;
        bool flag_df2;
        bool flag_df3;
task void assign_Workload();
  command result_t StdControl.init() {
        flag_df1=FALSE;
        flag_df2=FALSE;
        flag_df3=FALSE;
    return call CommControl.init();
  }

//Starts the application after 1 sec
  command result_t StdControl.start() {
        call Timer.start(TIMER_ONE_SHOT, 1000);
  }

  command result_t StdControl.stop() {
   call Timer.stop();
  return SUCCESS;
  }
  /**
   * Read sensor data in response to the <code>Timer.fired</code> event.
   *
   * @return The value of calling <tt>ADC.getData()</tt>.
   **/
   // To send a broadcast message
  event result_t Timer.fired() {
  MyBrMsg *MyData = (MyBrMsg *)pkt.data;

        if((call Send.send(0xFF, sizeof(MyBrMsg), &pkt)) == SUCCESS)
        {               call Leds.yellowOn();
        }
  }
event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
  {
        return SUCCESS;
```

**MyBaseStationM.nc**

```
}

//This event receives the messages from datafusion node and make sure gets back values
from all Datafusion nodes

event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {
        if(m->type==20)
        {
                if(m->addr==2)
                {
                 Df_Bs_ReceiveIntraVoltageMsg* Df_Bs_ReceiveIntraVoltageData =
(Df_Bs_ReceiveIntraVoltageMsg *)m->data;
                 volt_df1=Df_Bs_ReceiveIntraVoltageData->intra_voltage;
                flag_df1=TRUE;
                }

                if(m->addr==3)
                {
                        Df_Bs_ReceiveIntraVoltageMsg*
Df_Bs_ReceiveIntraVoltageData = (Df_Bs_ReceiveIntraVoltageMsg *)m->data;
                        volt_df2=Df_Bs_ReceiveIntraVoltageData->intra_voltage;
                        flag_df2=TRUE;
                }
                if(m->addr==4)
                {
                        Df_Bs_ReceiveIntraVoltageMsg*
Df_Bs_ReceiveIntraVoltageData = (Df_Bs_ReceiveIntraVoltageMsg *)m->data;
                        volt_df3=Df_Bs_ReceiveIntraVoltageData->intra_voltage;
                        flag_df3=TRUE;
                }

        }
                post assign_Workload();



                return m;
}
```

**MyBaseStationM.nc**

```
//This function allocates the task to the datafusion nodes based on the scheduling
algorithm

task void assign_Workload()
{
        Bs_Df_InterClusterMsg *Bs_Df_InterClusterData = (Bs_Df_InterClusterMsg
*)pkt.data;
                if(flag_df1 && flag_df2&& flag_df3)
                {
                        if(volt_df1>volt_df2)
                        {
                                if (volt_df1 < volt_df3)
                                {
                                        if (volt_df2 < volt_df3)
                                        {
                                        inter_p1=2;
                                        inter_p2=3;
                                        inter_p3=4;

                                        }
                                        else{
                                        inter_p1=2;
                                        inter_p2=4;
                                        inter_p3=3;

                                        }
                                }
                                else
                                {
                                        inter_p1=4;
                                        inter_p2=2;
                                        inter_p3=3;
                                }
                        }
                        else
                        {
                                if (volt_df2 < volt_df3)
                                {
                                        if (volt_df1 < volt_df3)
                                        {
```

**MyBaseStationM.nc**

```
                                inter_p1=3;
                                inter_p2=2;
                                inter_p3=4;
                                }
                                else
                                {
                                inter_p1=3;
                                inter_p2=4;
                                inter_p3=2;
                                }
                        }
                        else
                        {
                                inter_p1=4;
                                inter_p2=3;
                                inter_p3=2;
                        }
                }
                flag_df1=FALSE;
                flag_df2=FALSE;
                flag_df3=FALSE;


                for(i=1;i<=total_nodes-1;i++)
                {
                        deno=deno+temph;
                        temph*=h;
                }

                deno=deno+1;            // calculating denominator for Inter
Cluster
                df_node1=(uint16_t)(total_workload/deno);  // calculating
workload for DF1

                temph=0.99;
                h=0.99;
                for(i=1;i<2-1;i++)
                {temph*=h;}
                df_node2=(uint16_t)temph*df_node1; // calculating workload for
DF3
```

**MyBaseStationM.nc**

```
                    temph=0.99;
                    h=0.99;
                    for(i=1;i<3-1;i++)
                    {temph*=h;}
                    df_node3=(uint16_t)temph*df_node1;// calculating workload for
DF3


                    Bs_Df_InterClusterData->node_id[0]=inter_p1;
                    Bs_Df_InterClusterData->no_readings[0]=df_node1;

                    Bs_Df_InterClusterData->node_id[1]=inter_p2;
                    Bs_Df_InterClusterData->no_readings[1]=df_node2;

                    Bs_Df_InterClusterData->node_id[2]=inter_p3;
                    Bs_Df_InterClusterData->no_readings[2]=df_node3;


                    if((call Send.send(0, sizeof(Bs_Df_InterClusterMsg), &pkt)) ==
SUCCESS)
                    {

                    }

                }
                else
                {

                }
}

event result_t Send1.sendDone(TOS_MsgPtr msg, result_t success)
 {
        return SUCCESS;
 }
        event result_t RadioControl.initDone() {

   return SUCCESS;
```

***MyBaseStationM.nc***

```
 }
          event result_t RadioControl.startDone() {
         return SUCCESS;
 }
         event result_t RadioControl.stopDone() {

   return SUCCESS;
 }



}
```

***MyBaseStation.nc***

```
/*** This is the configuration file for the basestation node****/
includes MyBrMsg; //message to get energy values of the clusters
includes Df_Bs_ReceiveIntraVoltageMsg; //message used for BS node to receive the
voltage values
includes Bs_Df_InterClusterMsg; //message to get the result from the data fusion node


configuration MyBaseStation {
// this module does not provide any interfaces
}
implementation
{
  components Main, MyBaseStationM, TimerC, GenericComm as Comm,
LedsC,CC2420RadioC;

  Main.StdControl -> TimerC;
  Main.StdControl -> Comm;
  Main.StdControl -> MyBaseStationM;


        MyBaseStationM.Send -> Comm.SendMsg[AM_BroadcastVoltage];
        MyBaseStationM.Send1 -> Comm.SendMsg[AM_Bs_Df_InterCluster];
        MyBaseStationM.Receive ->
Comm.ReceiveMsg[AM_Df_Bs_ReceiveIntraVoltage];


        MyBaseStationM.Timer -> TimerC.Timer[unique("Timer")];


        MyBaseStationM.Leds ->LedsC;
        MyBaseStationM.CommControl -> Comm;
        MyBaseStationM.RadioControl->CC2420RadioC;


}
```

***MyMsg.h***

***Header File***

```
/***** This file is a header for sending a broadcastmessage****/
typedef struct MyBrMsg{
        uint8_t sendnumber;

}MyBrMsg;

enum {
        AM_BroadcastVoltage = 100
};
```

***Bs_Df_InterClusterMsg.h***

***Header File***

```
/***** This file is a header for sending a task allocation to datafusion nods*****/

typedef struct Bs_Df_InterClusterMsg{

        uint16_t node_id[3];
        uint16_t no_readings[3];
        uinit16_t value //single value or multi values
}Bs_Df_InterClusterMsg;


enum {
        AM_Bs_Df_InterCluster = 30
};
```

***Df_Bs_ReceiveIntraVoltageMsg.h***

***Df_Bs_ReceiveIntraVoltageMsg.h Header File***

```
/***** This file is a header for receiveing voltage of clusters from datafusion
nodes*****/

typedef struct Df_Bs_ReceiveIntraVoltageMsg{

        uint16_t intra_voltage;
}Df_Bs_ReceiveIntraVoltageMsg;
```

***Df_Bs_ReceiveIntraVoltageMsg.h***

```
enum {
        AM_Df_Bs_ReceiveIntraVoltage = 20
};
```

***DATAFUSION NODE***

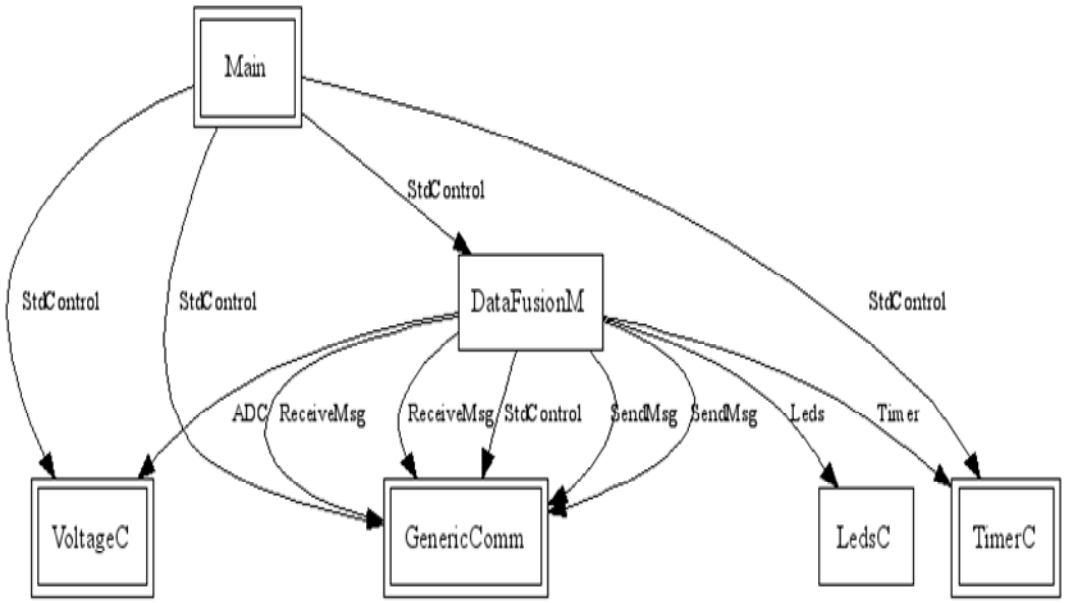


**Figure 6: Datafusion node Components Wring**

***DatafFusionM.nc***

```
/*********************************************
This program acts the data fusion node. it receives the voltage from the sensing node
calculates the clusters message and  sends the voltage of the cluster to the Base station.
Then receives messages from the Base Station about task and in turn assigns the task of
each sensing node
*********************************************/
includes Sn_DF_ReceiveVoltageMsg; //to receive voltage values from Datafusion node
includes Df_Bs_IntraVoltageMsg; //To send voltage values to intracluster message
includes Bs_Df_ReciveClusterTaskMsg; // To receive sensed data from sensing node
includes Df_Sn_IntraClusterMsg; // To assign task allocation to sensing node
module DataFusionM {
  provides {
    interface StdControl;
  }
  uses {
        interface Timer;
    //interface ADC;
        interface StdControl as CommControl;
```

**DatafFusionM.nc**

```
        interface SendMsg as Send;
        interface SendMsg as Send1; //AM_Df_Sn_IntraCluster
        interface ReceiveMsg as Receive;
        interface ReceiveMsg as Receive1; //Bs_Df_ReciveClusterTaskMsg
        interface Leds;
        interface ADC as voltage;
  }
}


implementation {

        TOS_Msg pkt;
        bool busy;
        uint16_t volt_sn1;
        uint16_t volt_sn2;
        uint16_t volt_curr_df;
        uint16_t intra_p1;
        uint16_t intra_p2;
        uint16_t tot_intra_volt;
        uint16_t no_read;
        uint16_t i=0;
        uint16_t total_nodes=2;

        float h=0.99;
        float temph=0.99;
        float deno=0;

        uint16_t sn_node1;
        uint16_t sn_node2;

        bool flag_sn1;

        bool flag_sn2;
        bool flag_Rec_task_BS;


        task void sendIntraVoltage();
```

**DatafFusionM.nc**

```
task void assign_task_to_sn();


  command result_t StdControl.init() {
        atomic busy = FALSE;
        atomic flag_sn1=FALSE;
        atomic flag_sn2=FALSE;
                return call CommControl.init();
  }


        event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {
        Sn_DF_ReceiveVoltageMsg* Sn_DF_ReceiveVoltageData =
(Sn_DF_ReceiveVoltageMsg *)m->data;
                if(m->type==10)
                {
                        if(m->addr==5)  //change this address depending on source
node's ID
                        {
                                volt_sn1=Sn_DF_ReceiveVoltageData->curr_voltage;
                                flag_sn1=TRUE;
                        }
                        else if (m->addr==6) //change this address depending on source
node's ID
                        {
                                volt_sn2=Sn_DF_ReceiveVoltageData->curr_voltage;
                                flag_sn2=TRUE;
                        }

                        if(flag_sn1 && flag_sn2)
                        {
                                flag_sn1=FALSE;
                                flag_sn2=FALSE;

                                //assign Priority to nodes based on voltage to Assign higher
payload
                                if(volt_sn1>volt_sn2)
                                {
                                        intra_p1=5;
```

**DatafFusionM.nc**

```
                                        intra_p2=6;
                                }
                                else
                                {
                                        intra_p1=6;
                                        intra_p2=5;
                                }
                                call voltage.getData();
                        }
                }
                return m;
        }


                //Get the voltage values when adc is ready
        async event result_t voltage.dataReady(uint16_t data) {
        volt_curr_df=0;
        volt_curr_df=data;
        post sendIntraVoltage();
        }



//Sends the voltage values to the basestation
        task void sendIntraVoltage()
        {
        Df_Bs_IntraVoltageMsg *Df_Bs_IntraVoltageData = (Df_Bs_IntraVoltageMsg
*)pkt.data;
        tot_intra_volt=volt_sn1+volt_sn2+volt_curr_df;
        Df_Bs_IntraVoltageData->intra_voltage=tot_intra_volt;

                if((call Send.send(0, sizeof(Df_Bs_IntraVoltageMsg), &pkt)) ==
SUCCESS) // change the value to hardcode the BS node id.
                {
                volt_sn1=0;
                volt_sn2=0;
                volt_curr_df=0;
                tot_intra_volt=0;
                        call Leds.yellowOn();
                }
```

```
        }


//On Receive messages from base station assign the task to the sensing node

event TOS_MsgPtr Receive1.receive(TOS_MsgPtr m) {

Bs_Df_ReciveClusterTaskMsg* Bs_Df_ReciveClusterTaskData =
(Bs_Df_ReciveClusterTaskMsg *)m->data;

                if(TOS_LOCAL_ADDRESS==Bs_Df_ReciveClusterTaskData-
>node_id[1])
                {
                no_read=Bs_Df_ReciveClusterTaskData->no_readings[1];
                flag_Rec_task_BS=TRUE;
                }

                if(TOS_LOCAL_ADDRESS==Bs_Df_ReciveClusterTaskData-
>node_id[2])
                {
                no_read=Bs_Df_ReciveClusterTaskData->no_readings[2];
                flag_Rec_task_BS=TRUE;
                }

                if(TOS_LOCAL_ADDRESS==Bs_Df_ReciveClusterTaskData-
>node_id[3])
                {
                no_read=Bs_Df_ReciveClusterTaskData->no_readings[3];
                flag_Rec_task_BS=TRUE;
                }


                //if DF node recieved the number of messages required..then assign
workload to sensing nodes..
                if(flag_Rec_task_BS)
                {
                        post assign_task_to_sn();
                }

return m;
```

**DatafFusionM.nc**

```
}


task void assign_task_to_sn()
{
        Df_Sn_IntraClusterMsg *Df_Sn_IntraClusterData = (Df_Sn_IntraClusterMsg
*)pkt.data;
                        for(i=1;i<=total_nodes-1;i++)
                        {
                                deno=deno+temph;
                                temph*=h;
                        }

                        deno=deno+1;            // calculating denominator for Intra
Cluster
                        sn_node1=(uint16_t)(no_read/deno);  // calculating workload for
Sensing node1

                        temph=0.99;
                        h=0.99;
                        for(i=1;i<2-1;i++)
                        {temph*=h;}
                        sn_node2=(uint16_t)temph*sn_node1; // calculating workload for
sn2 within the Datafusion nodes's cluster

                Df_Sn_IntraClusterData->node_id[0]=intra_p1;
                        Df_Sn_IntraClusterData->no_readings[0]=sn_node1;

                        Df_Sn_IntraClusterData->node_id[1]=intra_p2;
                        Df_Sn_IntraClusterData->no_readings[1]=sn_node2;

                if((call Send.send(0, sizeof(Df_Sn_IntraClusterMsg), &pkt)) ==
SUCCESS) // change the value to hardcode the SN node id.
                {

                }


}
```

**DatafFusionM.nc**

```
event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
 {
        return SUCCESS;
 }

event result_t Send1.sendDone(TOS_MsgPtr msg, result_t success)
 {
        return SUCCESS;
 }


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

 command result_t StdControl.stop() {

  return call Timer.stop();
 }

 event result_t Timer.fired() {

         return SUCCESS;

 }

  }
```

**DatafFusion.nc**

```
/*********************************************
This program acts the data fusion node.
it receives the voltage from the sensing node
calculates the clusters message and
sends the voltage of the cluster to the Base station.
Then receives messages from the Base Station about task
and inturn assigns the task of each sensing node
*********************************************/

includes Sn_DF_ReceiveVoltageMsg; //to receive voltage values from Datafusion node
includes Df_Bs_IntraVoltageMsg; //To send voltage values to intracluster message
includes Bs_Df_ReciveClusterTaskMsg; // To receive sensed data from sensing node
includes Df_Sn_IntraClusterMsg; // To assign task allocation to sensing node

configuration DataFusion {
// this module does not provide any interfaces
}
implementation
{
  components Main, DataFusionM, TimerC, GenericComm as Comm, LedsC,VoltageC;
  components new TimerMilliC() as ReportTime;

  Main.StdControl -> TimerC;
  //Main.StdControl -> Sensor;
  Main.StdControl -> Comm;
  Main.StdControl -> VoltageC;   //for voltage
  Main.StdControl -> DataFusionM;

        DataFusionM.Send -> Comm.SendMsg[AM_Df_Bs_IntraVoltage];
        DataFusionM.Send1 -> Comm.SendMsg[AM_Df_Sn_IntraCluster];
        DataFusionM.Receive -> Comm.ReceiveMsg[AM_Sn_DF_ReceiveVoltage];
        DataFusionM.Receive1 -> Comm.ReceiveMsg[AM_Bs_Df_ReciveClusterTask];

  DataFusionM.Timer -> TimerC.Timer[unique("Timer")];
  DataFusionM.Leds ->LedsC;
  DataFusionM.CommControl -> Comm;
  DataFusionM.voltage -> VoltageC;

}
```

***Header Files***

***Df_Bs_IntraVoltageMsg.h***

```
/**** This file structure is used to send voltage values from Data fusion node to Base station****/
typedef struct Df_Bs_IntraVoltageMsg{

        uint16_t intra_voltage;

}Df_Bs_IntraVoltageMsg;


enum {
        AM_Df_Bs_IntraVoltage = 20
};
```

***Bs_Df_ReciveClusterTaskMsg.h***

```
/**** This file structure is used to send task allocation from base station  node to Data Fusion node ****/

typedef struct Bs_Df_ReciveClusterTaskMsg{
        uint16_t node_id[3];
        uint16_t no_readings[3];
        uint16_t value //single or multi
}Bs_Df_ReciveClusterTaskMsg;

enum {
        AM_Bs_Df_ReciveClusterTask = 30
};
```

***Df_Sn_IntraClusterMsg.h***

***Df_Sn_IntraClusterMsg.h Header File***

```
/****This file structure is used to send task allocation to sensing node from data fusion node ***/
typedef struct Df_Sn_IntraClusterMsg{

        uint16_t node_id[2];
        uint16_t no_readings[2];
```

***Df_Sn_IntraClusterMsg.h***

```
}Df_Sn_IntraClusterMsg;
enum {
        AM_Df_Sn_IntraCluster = 40
};
```

***Sn_DF_ReceiveVoltageMsg.h***

***Header File***

```
/***This structure is used to receive voltage values from sensing node to datafusion
node*****/
typedef struct Sn_DF_ReceiveVoltageMsg{

        uint16_t curr_voltage;

}Sn_DF_ReceiveVoltageMsg;


enum {
        AM_Sn_DF_ReceiveVoltage = 10
};
```

***Source Node***

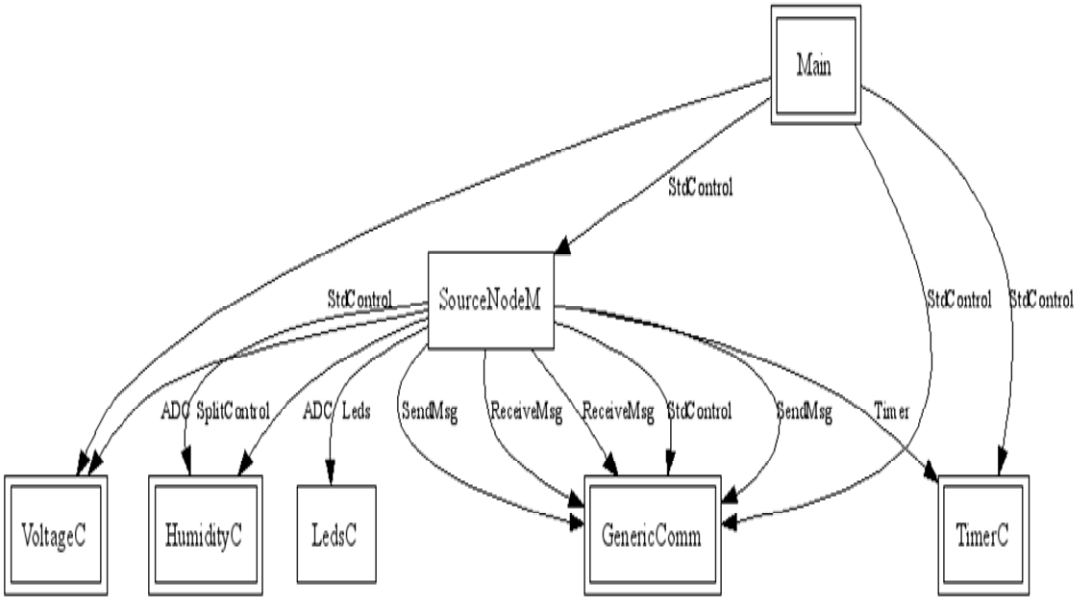


**Figure 7: Source Node Components Wiring**

***SourcenodeM.nc***

```
/************
This program acts as the source node. Initially it receives the broadcast message and
sends the voltage values to the datafusion node. Then completes the assigned task from
datafusion node and sends the results back to the data fusion node.
************/
includes bs_sn_broadcast; //Message to receive the brocast message
includes Sn_Df_powerMsg; //message to send the voltage values
includes Df_Sn_ReciveClusterTaskMsg; //message to receive the task allocation msg
includes Sn_Df_dataMsg;//message to send the results back to the data fusion node
module SourceNodeM {
 provides {
   interface StdControl;
 }
 uses {
        interface Timer;
   interface ADC;
        interface StdControl as CommControl;
        interface SendMsg as Send;
```

**SourcenodeM.nc**

```
        interface SendMsg as Send1;

interface ReceiveMsg as Receive;
        interface ReceiveMsg as Receive1;//AM_Df_Sn_ReciveClusterTask
        interface Leds;
        interface SplitControl as TemperatureControl;
        interface ADC as voltage; //for voltage
        interface ADC as SensorTemp;


  }
}


implementation {

        //Messages
        TOS_Msg pkt;


        //Variables
        uint16_t voltage_read;
        uint16_t no_read;
        uint16_t id_node;
        uint16_t packetsSent=0;
        uint16_t readings[500];
        uint8_t tempsend=0;

        uint16_t i=0;
        uint16_t j=0;
        uint16_t k=0;
        bool busy;

        //Tasks
        task void sendVoltage_DFnode();
        task void temp();
        task void mysend();

  command result_t StdControl.init() {
```

**SourcenodeM.nc**

```
        call TemperatureControl.init();
        call TemperatureControl.start();
  return call CommControl.init();
 }

 command result_t StdControl.start() {
            return call Timer.start(TIMER_ONE_SHOT, 2000);
 }

 command result_t StdControl.stop() {

  return call Timer.stop();
 }

 event result_t Timer.fired() {
        return SUCCESS;
 }

  async event result_t ADC.dataReady(uint16_t data) {

 }



/**Receives the Broadcast Message from BaseStation(BS) and sends the Voltage value to
the
Data Fusion Node (DF).
**/
        event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {
        call Leds.greenOn();
                if(m->type==100)
                {
                call Leds.redOn();
                call voltage.getData();
                }
                return m;
        }


/**Receives the assigned task from DataFusionnode(DF) and sends the temp value to the
```

**SourcenodeM.nc**

```
Data Fusion Node (DF).
**/
        event TOS_MsgPtr Receive1.receive(TOS_MsgPtr m) {
        Df_Sn_ReciveClusterTaskMsg* Df_Sn_ReciveClusterTaskData =
(Df_Sn_ReciveClusterTaskMsg *)m->data;

                if(m->type==40)
                {
                        //if df node id =2 then sn nodes id are 5 and 6
                        if(m->addr==2)
                        {

                                if(TOS_LOCAL_ADDRESS==5)
                                {
                                        id_node=Df_Sn_ReciveClusterTaskData-
>node_id[0];
                                        no_read=Df_Sn_ReciveClusterTaskData-
>no_readings[0];
                                        call ADC.getData();
                                }

                                if(TOS_LOCAL_ADDRESS==6)
                                {
                                        id_node=Df_Sn_ReciveClusterTaskData-
>node_id[1];
                                        no_read=Df_Sn_ReciveClusterTaskData-
>no_readings[1];
                                        call ADC.getData();
                                }

                        }

                //if df node id =3 then sn nodes id are 7 and 8
                        if(m->addr==3)
                        {

                                if(TOS_LOCAL_ADDRESS==7)
                                {
                                        id_node=Df_Sn_ReciveClusterTaskData-
>node_id[0];
```

**SourcenodeM.nc**

```
                                        no_read=Df_Sn_ReciveClusterTaskData-
>no_readings[0];
                                        call ADC.getData();
                                }

                                if(TOS_LOCAL_ADDRESS==8)
                                {
                                        id_node=Df_Sn_ReciveClusterTaskData-
>node_id[1];
                                        no_read=Df_Sn_ReciveClusterTaskData-
>no_readings[1];
                                        call ADC.getData();
                                }

                        }

                        //if df node id =4 then sn nodes id are 9 and 10
                        if(m->addr==4)
                        {

                                if(TOS_LOCAL_ADDRESS==9)
                                {
                                        id_node=Df_Sn_ReciveClusterTaskData-
>node_id[0];
                                        no_read=Df_Sn_ReciveClusterTaskData-
>no_readings[0];
                                        call ADC.getData();
                                }

                                if(TOS_LOCAL_ADDRESS==10)
                                {
                                        id_node=Df_Sn_ReciveClusterTaskData-
>node_id[1];
                                        no_read=Df_Sn_ReciveClusterTaskData-
>no_readings[1];
                                        call ADC.getData();
                                }

                        }
```

```
                }
                return m;

        }



/**
 * In response to <code>ADC.dataReady</code>, store sensor data
 * and post task for averaging.
 * @return returns <code>SUCCESS</code>
 **/


async event result_t SensorTemp.dataReady(uint16_t data) {

        readings[packetsSent]=data;
        atomic packetsSent++;

        if(packetsSent==no_read)
        {
        Sn_Df_dataMsg *Sn_Df_data = (Sn_Df_dataMsg *)pkt.data;

                        for(i=0,j=0;i<10;i++,j++)
                        Sn_Df_data->read[i]=readings[i];
                        if(!busy)
                        {
                                post mysend();

                        }
        }
        else
        return call SensorTemp.getData();

        //return SUCCESS;
}
```

**SourcenodeM.nc**

```
task void temp()
{
Sn_Df_dataMsg *Sn_Df_data = (Sn_Df_dataMsg *)pkt.data;

        if(!busy);
        {
                for(i=0;i<50;i++)
                {
                        Sn_Df_data->read[i]=readings[j];
                        j++;
                }

                post mysend();
        }

}

//send the data back to the Data Fusion node
task void mysend()
{
        Sn_Df_dataMsg *Sn_Df_data = (Sn_Df_dataMsg *)pkt.data;

        tempsend=tempsend+1;
        Sn_Df_data->node_id=id_node;
        if((call Send1.send(0, sizeof(Sn_Df_dataMsg), &pkt)) == SUCCESS)
        {
                atomic busy = TRUE;

        }


}


//Get the voltage values when adc is ready
 async event result_t voltage.dataReady(uint16_t data)
{
        voltage_read=0;
        voltage_read=data;
        call Leds.greenOn();
```

**SourcenodeM.nc**

```
        post sendVoltage_DFnode();
 }

        //send the Voltage values to the datafusion nodes.
        task void sendVoltage_DFnode()
        {
                Sn_Df_powerMsg *Sn_Df_powerData = (Sn_Df_powerMsg *)pkt.data;
                Sn_Df_powerData->curr_voltage=voltage_read;
                if((call Send.send(0, sizeof(Sn_Df_powerMsg), &pkt)) == SUCCESS) //
change the value to hardcode the DF node id.
                {
                        call Leds.yellowOn();
                }

        }

        event result_t Send.sendDone(TOS_MsgPtr msg, result_t success){
        return SUCCESS;
 }


        event result_t Send1.sendDone(TOS_MsgPtr msg, result_t success){
        return SUCCESS;
 }
        event result_t TemperatureControl.initDone() {

        return SUCCESS;
 }

         event result_t TemperatureControl.startDone() {

        //call ReportTime.startOneShot(1024);
   return SUCCESS;
 }
        event result_t TemperatureControl.stopDone() {
   return SUCCESS;
 }
```

**Sourcenode.nc**

```
/************
This program acts as the source node. Initially receieves the broadcast message and
sends the voltage values to the
datafusion node. Then completes the assigned task from datafusion node and sends the
results back to the
data fusion node.
************/

includes bs_sn_broadcast; //Message to receive the brocast message
includes Sn_Df_powerMsg; //message to send the voltage values
includes Df_Sn_ReciveClusterTaskMsg; //message to receive the task allocation
messages
includes Sn_Df_dataMsg;//message to send the results back to the data fusion node


configuration SourceNode {
// this module does not provide any interfaces
}
implementation
{
  components Main, SourceNodeM, TimerC, HumidityC as SensorTemp, GenericComm
as Comm, LedsC,VoltageC;


  Main.StdControl -> TimerC;

  Main.StdControl -> Comm;
  Main.StdControl -> SourceNodeM;
  Main.StdControl -> VoltageC;   //for voltage
        SourceNodeM.Send -> Comm.SendMsg[AM_Sn_Df_power];
        SourceNodeM.Send1 -> Comm.SendMsg[AM_Sn_Df_Data];
        SourceNodeM.Receive -> Comm.ReceiveMsg[AM_Bs_Sn_Broadcast];
        SourceNodeM.Receive1 -> Comm.ReceiveMsg[AM_Df_Sn_ReciveClusterTask];
  SourceNodeM.Timer -> TimerC.Timer[unique("Timer")];
  SourceNodeM.ADC -> SensorTemp.Temperature;
  SourceNodeM.Leds ->LedsC;

  SourceNodeM.CommControl -> Comm;
  SourceNodeM.TemperatureControl -> SensorTemp;
```

**Sourcenode.nc**

```
  SourceNodeM.voltage -> VoltageC;

}
```

***Header Files***

**Bs_Sn_BroadcastMsg.h**

***Header File***

```
/***** This structure is used to receive message from base station*******/
typedef struct Bs_Sn_BroadcastMsg{

        uint8_t sendnumber;

}Bs_Sn_BroadcastMsg;


enum {
        AM_Bs_Sn_Broadcast = 100
};
```

**Df_Sn_ReciveClusterTaskMsg.h**

***Header File***

```
/***** This structure is used to receive message from Data fusion node about task
allocation*******/
typedef struct Df_Sn_ReciveClusterTaskMsg{
;
        uint16_t node_id[2];
        uint16_t no_readings[2];

}Df_Sn_ReciveClusterTaskMsg;


enum {
        AM_Df_Sn_ReciveClusterTask = 40
};
```

***bs_sn_broadcast.h***

***Header File***

```
/***** This structure is used to send broadcast message to base station*******/

typedef struct bs_sn_broadcast{

        uint8_t sendnumber;

}bs_sn_broadcast;

enum {
        AM_bs_sn_broadcaset = 10
};
```

***Sn_Df_dataMsg.h***

***Header File***

```
/*******This structure is used to send data to the datafusion node from sensor
node*****/
typedef struct Sn_Df_dataMsg{

        uint16_t read[10];
        uint16_t node_id;

}Sn_Df_dataMsg;


enum {
        AM_Sn_Df_Data = 50
};
```

***Sn_Df_powerMsg.h***

***Header File***

```
/*******Structure is used to send the currentvoltage value to Datafusion Node******/
```

***Sn_Df_powerMsg.h***

```
typedef struct Sn_Df_powerMsg{

        uint16_t curr_voltage;

}Sn_Df_powerMsg;


enum {
        AM_Sn_Df_power = 10
};
```

# Appendix B Compression Code:

***List of Files***

- *CompressionM.nc*
- *Compression.nc*
- *MyMsg.h*
- *Make File*

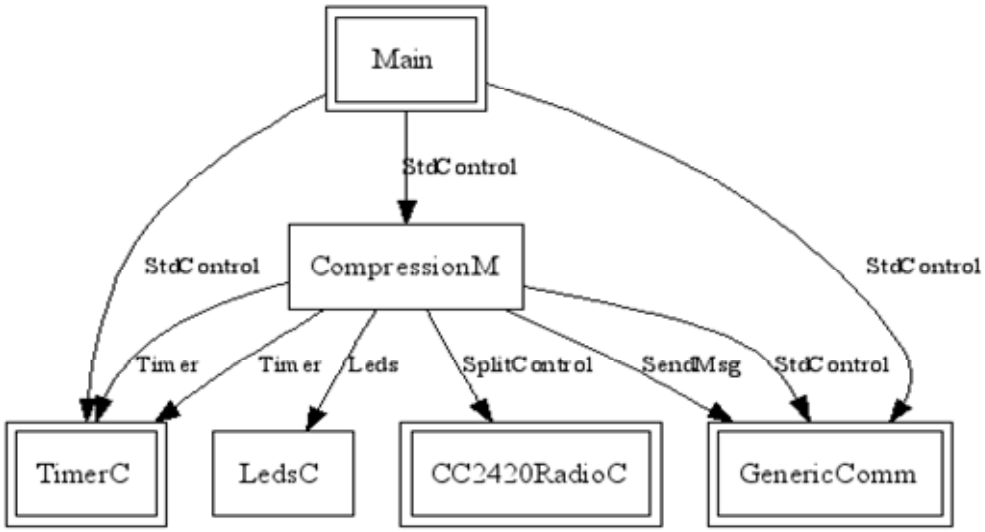


**Figure 8: Compression Components Wiring**

***CompressionM.nc***

```
includes MyMsg; // to send the encoded string to other nodes

module CompressionM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
        interface Timer as Timer2;

        interface StdControl as CommControl;
        interface SendMsg as Send;
        interface Leds;
```

**CompressionM.nc**

```
interface SplitControl as RadioControl;

  }
}

implementation {
        //TOS_Msg pkt;
        //uuint16_t16_t i;

// ------------------------------------------------------------------------------------------
//structure to build the tree for Huffman encoding
struct code
        {
        char ch;
        uint16_t fn;
        char * hcode;
        uint16_t l_hcode;
        struct code * prev;
        struct code * next;
        struct code * left;
        struct code * right;
        };

union long2char
        {
        uint32_t byte;
        uint8_t bb[4];
        };

union int2char
        {
        uint16_t byte;
        uint8_t bb[2];
        };

char ddd[300];
uint16_t ddt=0;
uint32_t tot_count=0;

// ------------------------------------------------------------------------------------------
```

**CompressionM.nc**

```
//function delcarations
uint16_t char_count(char * src,uint16_t src_length, char ** chs,uint16_t ** chc);
uint8_t * compress_string(char * source_string, uint16_t src_length, uint16_t *
ret_enc_length);
struct code * greedy_huffman(char * chs,uint16_t * chc,uint16_t cc);
struct code * make_codes(struct code * head, uint16_t cc);
uint8_t * encode_string(char * src,uint16_t src_length, struct code * table, uint16_t cc,
uint16_t * enc_length);
uint16_t copy_bytes(char * trgt,uint16_t cur_index,uint8_t * src,uint8_t nbytes);
uint16_t free_tree(struct code * root);
char * decode_string(char * src, uint16_t * dcd_length);


//Function which converts the sourcestring into encoded compress string
uint8_t * compress_string(char * source_string, uint16_t src_length, uint16_t *
ret_enc_length)
{
char * chs=NULL;
uint16_t * chc=NULL;
uint16_t ccount;
uint8_t * encoded_string;
struct code * head=NULL;
struct code * table=NULL;
uint16_t enc_length;

ccount = char_count(source_string,src_length,&chs,&chc);                    // find
frequency spectrum of characters ...
head = greedy_huffman(chs,chc,ccount);
table=make_codes(head,ccount);
encoded_string = encode_string(source_string, src_length, table,ccount,&enc_length);
*ret_enc_length = enc_length;

if (table!=NULL) free(table);
if (chs!=NULL) free(chs);
if (chc!=NULL) free(chc);
if (head!=NULL) free_tree(head);

return(encoded_string);
}
```

**CompressionM.nc**

```
//Function to decode the sourcestring using the decoded length
char * decode_string(char * src, uint16_t * dcd_length)
{
uint16_t ccd=0;
//char ch1,ch2;
char ch;
char ccc[30];
uint32_t nc;
uint16_t n1;
uint16_t ns=0;
struct code *head,*tt,*tt1;
uint16_t i,j;
uint16_t k;
uint16_t ttp[8]={128,64,32,16,8,4,2,1};
char bb[8];
uint16_t nbits;
union long2char lc1;
union int2char ic1;
uint16_t src_index = 0;
uint16_t tgt_index = 0;
char * tgt;

for(i=0;i<4;i++) {lc1.bb[i] = src[src_index];src_index++;}
nc = lc1.byte;
for(i=0;i<2;i++) {ic1.bb[i] = src[src_index];src_index++;}
ns = ic1.byte;

tgt = (char *)malloc(sizeof(char)*(nc+1));
head=(struct code *)malloc(sizeof(struct code));
head->left=NULL;
head->right=NULL;
head->ch='+';
for(i=0;i<ns;i++)
        {
        ch = src[src_index];src_index++;
        for(i=0;i<2;i++) {ic1.bb[i] = src[src_index];src_index++;}
        n1 = ic1.byte;
        for(j=0;j<n1;j++) {ccc[j]=src[src_index];src_index++;}
        tt = head;
        for(j=0;j<n1;j++)
```

```
                {
                if (ccc[j]=='0')
                        {
                        if (tt->left==NULL)
                                {
                                tt1=(struct code *)malloc(sizeof(struct code));
                                tt1->left=NULL;
                                tt1->right=NULL;
                                tt1->ch='+';
                                tt->left=tt1;
                                }
                        tt=tt->left;
                        }
                if (ccc[j]=='1')
                        {
                        if (tt->right==NULL)
                                {
                                tt1=(struct code *)malloc(sizeof(struct code));
                                tt1->left=NULL;
                                tt1->right=NULL;
                                tt1->ch='+';
                                tt->right=tt1;
                                }
                        tt=tt->right;
                        }
                }
        tt->ch=ch;
        }

nbits=0;
tt=head;
i=0;
while(i<nc)
        {
        if (nbits==8)
                {
                ch = src[src_index];src_index++;
                k=(uint16_t)ch;
                for(j=0;j<8;j++)
                        {
```

```
                        if ((k&ttp[j])!=0) bb[j]='1'; else bb[j]='0';
                        }
                nbits=0;
                }
        while(nbits<8)
                {
                if ((tt->left==NULL)&&(tt->right==NULL))
                        {
                        ch=tt->ch;
                        tgt[tgt_index]=ch;tgt_index++;
                        tt=head;
                        i++;
                        if (i>=nc) break;
                        }
                if (bb[nbits]=='0') tt=tt->left;
                if (bb[nbits]=='1') tt=tt->right;
                nbits++;
                }
        }

*dcd_length = nc;
return(tgt);
}

uint8_t * encode_string(char * src,uint16_t src_length, struct code * table, uint16_t cc,
uint16_t * enc_length)
{
union long2char lc1;
union int2char ic1;

char ch,ch1;
uint16_t nn;
uint16_t index[256];
uint16_t i,j,k;
uint8_t xx;
uint16_t nbits;
char * trs;
uint16_t trs_index=0;
uint16_t src_index=0;
```

```
if (src_length<1) {*enc_length = -1;return(NULL);}
if (table==NULL) {*enc_length = -2;return(NULL);}
if (cc<1) {*enc_length = -3;return(NULL);}

for(i=0;i<256;i++) index[i]=-1;
for(i=0;i<cc;i++)
        {
        j = (uint16_t)table[i].ch;
        index[j]=i;
        }

trs = (uint8_t *)malloc(sizeof(uint8_t)*(4+4*cc+tot_count+2));

lc1.byte = tot_count;trs_index = copy_bytes(trs,trs_index,lc1.bb,4);
ic1.byte = cc;trs_index = copy_bytes(trs,trs_index,ic1.bb,2);

for(i=0;i<cc;i++)
        {
        ch1 = table[i].ch;
        trs[trs_index] = (uint8_t)ch1;trs_index++;
        nn = table[i].l_hcode;
        ic1.byte = nn;trs_index = copy_bytes(trs,trs_index,ic1.bb,2);
        for(j=0;j<nn;j++)
                {
                ch1 = table[i].hcode[j];
                trs[trs_index] = (uint8_t)ch1;trs_index++;
                }
        }

xx=0;
nbits=0;
while(src_index<src_length)
        {
        ch = src[src_index];src_index++;
        j = (uint16_t)ch;
        k = index[j];
        for(i=0;i<table[k].l_hcode;i++)
                {
                xx=xx+(uint8_t)(table[k].hcode[i]-'0');
```

```
                nbits++;
                if (nbits==8)
                        {
                        nbits=0;
                        trs[trs_index] = xx;trs_index++;
                        xx=0;
                        }
                else
                        {
                        xx=xx<<1;
                        }
                }
        }
if (nbits!=0)
        {
        for(i=nbits+1;i<8;i++) xx=xx<<1;
        trs[trs_index] = xx;trs_index++;
        }

*enc_length = (trs_index - 1);
return(trs);
}



struct code * make_codes(struct code * head, uint16_t cc)
{
struct code *table, *curr;
struct code * stack[300];
uint16_t stack_count=0;
uint16_t done=0;
uint16_t i;
char hcc[300];
uint16_t lcd[300];
uint16_t hcd=0;
uint16_t tc=0;
uint16_t rs=0;

if (cc<=0) return(NULL);
table = (struct code *)malloc(sizeof(struct code)*cc);
```

```
if (cc==1)
        {
        table[0].l_hcode=1;
        table[0].hcode=malloc(sizeof(char)*1);
        table[0].hcode[0]=head->hcode[0];
        table[0].ch = head->ch;
        return(table);
        }

rs=0;
curr = head;

while(!done)
        {
        if (curr->right!=NULL)
                {
                stack_count++;
                stack[stack_count]=curr->right;
                lcd[stack_count]=hcd;
                }

        if (curr->left!=NULL)
                {
                curr=curr->left;
                hcc[hcd]='0';
                hcd=hcd+1;
                continue;
                }

        if ((curr->left==NULL)&&(curr->right==NULL))
                {
                table[tc].ch = curr->ch;
                table[tc].l_hcode = hcd;
                table[tc].hcode = (char *)malloc(sizeof(char)*hcd);
                for(i=0;i<hcd;i++) table[tc].hcode[i]=hcc[i];
                tc++;

                if (stack_count!=0)
                        {
```

```
                        rs=1;
                        }
                else
                        {
                        done=1;
                        break;
                        }
                }

        if (rs==1)
                {
                curr=stack[stack_count];
                hcd = lcd[stack_count];
                hcc[hcd]='1';
                hcd++;
                stack_count--;
                }
        }

return(table);
}

struct code * greedy_huffman(char * chs,uint16_t * chc,uint16_t cc)
{
uint16_t i;
struct code * ll;
struct code *head,*tail,*tt1,*tt2,*tt3;
uint16_t count;
uint16_t m1,m2,mf=0;
uint16_t done=0;

if (cc<=0) return(NULL);

if (cc==1)
        {
        ll = (struct code *)malloc(sizeof(struct code)*1);
        ll[0].ch = chs[0];
        ll[0].fn = chc[0];
        ll[0].hcode = (char *)malloc(sizeof(char)*1);
        ll[0].hcode[0] = '0';
```

```
        ll[0].l_hcode = 1;
        ll[0].left=NULL;
        ll[0].right=NULL;
        ll[0].prev=NULL;
        ll[0].next=NULL;
        return(ll);
        }

for(i=0;i<cc;i++)
        {
        ll = (struct code *)malloc(sizeof(struct code)*1);
        if (mf<chc[i]) mf=chc[i];
        ll->ch = chs[i];
        ll->fn = chc[i];
        ll->l_hcode = 0;
        ll->hcode = NULL;
        ll->left = NULL;
        ll->right = NULL;
        ll->next = NULL;
   if (i==0)
                {
                head=ll;
                head->prev = NULL;
                tail=head;
                }
        else
                {
                tail->next = ll;
                ll->prev = tail;
                tail=ll;
                }
        }

count = cc;
mf++;
while(head->next!=NULL)
        {
        m1=head->fn;
        tt1=head;
        ll=head;
```

```
done=0;
while(!done)
        {
        if (m1>ll->fn)
                {
                tt1=ll;
                m1=ll->fn;
                }
        if (ll->next==NULL) done=1; else ll=ll->next;
        }

if (tt1->prev!=NULL) {tt1->prev->next=tt1->next;}
else
        {
        head=tt1->next;
        if (head!=NULL) head->prev=NULL;
        }

if (tt1->next!=NULL) {tt1->next->prev=tt1->prev;}

m2=head->fn;
tt2=head;
ll=head;
done=0;
while(!done)
        {
        if (m2>ll->fn)
                {
                tt2=ll;
                m2=ll->fn;
                }
        if (ll->next==NULL) done=1; else ll=ll->next;
        }

if (tt2->prev!=NULL) {tt2->prev->next=tt2->next;}
else
        {
        head=tt2->next;
        if (head!=NULL) head->prev=NULL;
        }
```

```
        if (tt2->next!=NULL) {tt2->next->prev=tt2->prev;}

        tt3 = (struct code *)malloc(sizeof(struct code)*1);
        tt3->ch = '+';
        tt3->fn = m1+m2;
        tt3->l_hcode = 0;
        tt3->hcode = NULL;
        tt3->next = NULL;
        tt3->prev = NULL;
        if (m1<=m2)
                {
                tt3->left = tt1;
                tt3->right = tt2;
                }
        else
                {
                tt3->left = tt2;
                tt3->right = tt1;
                }

        if (head==NULL)
                {
                head=tt3;
                }
        else
                {
                for(ll=head;ll->next!=NULL;ll=ll->next);
                ll->next=tt3;
                tt3->prev=ll;
                }
        }

ll=head;
return(ll);
}

uint16_t char_count(char * src,uint16_t src_length, char ** chs,uint16_t ** chc)
{
uint32_t src_index=0;
```

```
uint16_t res=0;
char * cha_l;
uint16_t * chc_l;
uint16_t xhc[256];
uint16_t i,j,count;

if (src_length<1)
        {
        *chs=NULL;
        *chc=NULL;
        return(-1);
        }

for(i=0;i<256;i++) xhc[i]=0;
while(src_index<src_length)
        {
        i=(uint16_t)src[src_index];
        src_index++;
        xhc[i]=xhc[i]+1;
        tot_count++;
        }

count=0;
for(i=0;i<256;i++) if (xhc[i]>0) count++;

chc_l = (uint16_t *)malloc(sizeof(uint16_t)*count);
cha_l = (char *)malloc(sizeof(char)*count);

j=0;
for(i=0;i<256;i++)
        {
        if (xhc[i]>0)
                {
                chc_l[j]=xhc[i];
                cha_l[j]=(char)i;
                j=j+1;
                }
        }

*chc = chc_l;
```

```
*chs = cha_l;
res=count;

return(res);
}

uint16_t free_tree(struct code * root)
{
if (root==NULL) return(0);
if (root->left!=NULL) free_tree(root->left);
if (root->right!=NULL) free_tree(root->right);
if (root->hcode!=NULL) free(root->hcode);
return(0);
}

uint16_t copy_bytes(char * trgt,uint16_t cur_index,uint8_t * src,uint8_t nbytes)
{
uint8_t i;

for(i=0;i<nbytes;i++)
        {
        trgt[cur_index]=src[i];
        cur_index++;
        }
return(cur_index);
}

// ------------------------------------------------------------------------------------------

void myfunction();

  command result_t StdControl.init() {

        //call RadioControl.init();


        return call CommControl.init();
  }

  /**
```

**CompressionM.nc**

```
 * Starts the timer.
 *
 * @return The value of calling <tt>Timer.start()</tt>.
 **/
command result_t StdControl.start() {

        call Timer2.start(TIMER_ONE_SHOT, 1000);
        return call Timer.start(TIMER_ONE_SHOT, 5000);

}


/**
 * Stops the timer.
 *
 * @return The value of calling <tt>Timer.stop()</tt>.
 **/
command result_t StdControl.stop() {
  call Timer2.stop();
      return call Timer.stop();
}

/**
 * Read sensor data in response to the <code>Timer.fired</code> event.
 *
 * @return The value of calling <tt>ADC.getData()</tt>.
 **/
event result_t Timer.fired() {

       myfunction();
}


      event result_t Timer2.fired() {


}

      void myfunction()
      {
```

**CompressionM.nc**

```
TOS_Msg pkt;
                MyMsg *MyData = (MyMsg *)pkt.data;
                uint16_t enc_len=0;
                uint16_t i=0;
                uint8_t * ss;
                uint8_t b[5];



                for(i=0;i<5;i++)
                {
                b[i]=2;
                }
                //b[30]='\0';

                ss=compress_string(b,5,&enc_len);
                for(i=0;i<enc_len;i++) MyData->a[i]=ss[i];
                        for(i=0;i<enc_len;i++) MyData->a[i]=ss[i];

                        if((call Send.send(0xffff, sizeof(MyMsg), &pkt)) == SUCCESS)
                        {

                        }
        }


 /**
  * In response to <code>ADC.dataReady</code>, store sensor data
  * and post task for averaging.
  * @return returns <code>SUCCESS</code>
  **/



event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
 {
 call Leds.greenToggle();
 //call RadioControl.stop();
        return SUCCESS;
```

***CompressionM.nc***

```
  }



        event result_t RadioControl.initDone() {


   return SUCCESS;
  }
        event result_t RadioControl.startDone() {
        call Leds.yellowOn();

        return SUCCESS;
  }

        event result_t RadioControl.stopDone() {
  call Leds.redOn();
   return SUCCESS;
  }

}
```

**Compression.nc**

```
includes MyMsg;

configuration Compression {
// this module does not provide any interfaces
}
implementation
{
  components Main, CompressionM, TimerC, GenericComm as Comm,
LedsC,CC2420RadioC;

  Main.StdControl -> TimerC;

  Main.StdControl -> Comm;
  Main.StdControl -> CompressionM;


        CompressionM.Send -> Comm.SendMsg[AM_STEST];


        CompressionM.Timer -> TimerC.Timer[unique("Timer")];
        CompressionM.Timer2 -> TimerC.Timer[unique("Timer")];
        CompressionM.Leds ->LedsC;

        CompressionM.CommControl -> Comm;
CompressionM.RadioControl->CC2420RadioC;
}
```

***MyMsg.h***

***Header Files***

```
//MyMsg,H Header file to send the encoded string
#include <string.h>
#include <stdlib.h>
typedef struct MyMsg{
        uint8_t a[100];
}MyMsg;
enum {
        AM_STEST = 55
};
```

***MyMsg1.h***

***Header file***

```
//To receie the actual encoded string
typedef struct MyMsg1{
        uint8_t a[100];\
}MyMsg1;
enum {
        AM_STEST1 = 66
};
```

***MyMsg.h***

***Header file***

```
//To receive the encoded string on the node
#include <string.h>
#include <stdlib.h>
typedef struct MyMsg{
        uint8_t a[100];

}MyMsg;
enum {
        AM_STEST = 55
};
```

# Appendix C Precision Agriculture Code

**Packet Format of Sensing Node**

| | | Bytes |
|---|---|---|
| uint16_t adc_zero[12]; | 2Bytes*12 | 24 |
| uint16_t adc_one[12]; | 2Bytes*12 | 24 |
| uint16_t adc_two[12]; | 2Bytes*12 | 24 |
| uint16_t adc_three[12]; | 2Bytes*12 | 24 |
| uint16_t volt; | 2Bytes | 2 |
| uint16_t node_id; | 2Bytes | 2 |
| | | 100 bytes |

adc_zero reports soil senors values from soilsensor 1 attached to a one sensing node(TMOTE)
adc_one reports soil senors values from soilsensor 2 attached to a one sensing node(TMOTE)
adc_two reports soil senors values from soilsensor 2 attached to a one sensing node(TMOTE)
adc_three reports soil senors values from soilsensor 2 attached to a one sensing node(TMOTE)
volt represents the voltage values the values will be either ff ff (when high) or o(when low)

**Data Storage Format**

Data is stored on the flash of the central node( ReceiveData Program)

Flash consists of 16 blocks
Each block has 256 pages
Each page has 256 bytes

One sensing node sends 100 bytes/day, this is stored in one page on the flash based on the node's id. Each sensing node's data (100 bytes) is stored in a single page.

One block can store 256 pages.We have used one block to store 10 node's data for 21 days.

Sampling frequency of each sensors (sensing node)
Each sensor node consists of four soil moisture sensor nodes. Each node samples all the four sensors every 2 hours and sends(every day i.e. after 24 hours and 12 readings from each sensor) the above mentioned data packet to the central node for storage on the flash of the central node.

Central node to the Base station
Base station program has to be installed on one tmote and attached to the computer.

The data from the central node can be retrieved by pressing the 'user button' on the tmote. The central node can be brought from the agricultural field any time, and the data can be retrieved.

Instructions :

**Sensing Node**
Install the soiltest program on all the nodes with the soil sensor moistures attached to it.

To **compile** the program
Go to the soiltest folder
From the program folder type

'Make tmote'

To **install** the program
The node id for the sensing nodes can be from 2. Please make sure the node id for the sensor nodes are not 0 and 1, because '0' is used for basestation node and node id '1' is used for central node.
'Make tmote reinstall,2'

**Central node**
Before installing the central node's program it is good to clear the node's flash.

/opt/tinyos-1.x/apps/TestNewFlash/Format
From this folder intsall the format program on a tmote
To compile type
'Make tmote'

To **install** the program type
Make tmote reinstall,1

After the program is installed on the node wait till the 'green Led' flashes on the node. The flashing of green leds means that the flash has been cleared.

Install the receivedata (central node) program on one node.

To compile: /opt/moteiv/apps/receiveData in this folder
Type 'Make tmote'

To **install** the program type
Make tmote reinstall,1

1 refers to the central node's node id. Sensing node's send the data to the node id '1'

Listen.java
This program converts the data from hex to integer and saves the data in text file in an orderly manner.

**Base station**
Before installing the base station program, please make sure the \cygwin\opt\tinyos-1.x\tools\java\net\tinyos\tools\Listen.java is installed

To compile
\cygwin\opt\tinyos-1.x\tools\java\net\tinyos\tools\
And type
'make' to compile the new java program.

To compile the base station progrm
Cd /opt/moteiv/apps/TOSBase in this folder
Type 'Make tmote'

To install the basestation program

Type ‘make tmote reinstall,0’

Type
‘Motelist’ to get the port no
The type

‘export MOTECOM=serial@COM<portno> :57600’

Then type
‘java net.tinyos.tools.Listen’

To retrieve the data (i.e. after pressing the ‘user button’ on the central node)

**Output Data Format**

The text files are stred on a folder under /opt/moteiv/SoilTestReadings
The text files are named based on the node id’s i.e. for sensing node with node id 2 the text file is named as ‘Mote2.txt’, ‘Mote3.txt’ etc…

| Sensor1 | Sensor2 | Sensor3 | Sensor4 | Time Packet Received |
|---|---|---|---|---|
| 0.498779 | 0.72161174 | 0.73992676 | 0.519536 | Fri Oct 1 (time) |
| 0.498779 | 0.72161174 | 0.73992676 | 0.519536 | Fri Oct 19 (time) |

The above lines shows an example of a text file the way the data are stored on a text file. Sensor1,sensor2,sensor3,sensor4 represents Node2’s data i.e. soil senor values.

Sensor1 corresponds to soil moisture1 connected to node2 on ADC0
Sensor2 corresponds to soil moisture2 connected to node2 on ADC1
Sensor3 corresponds to soil moisture3 connected to node2 on ADC2
Sensor4 corresponds to soil moisture4 connected to node2 on ADC3

***Soil Moisture Node code (Sensing Node with soil moisture sensor)***

***List of Files***

- *soilTestM.nc*
- *soilTest.nc*
- *MyMsg*

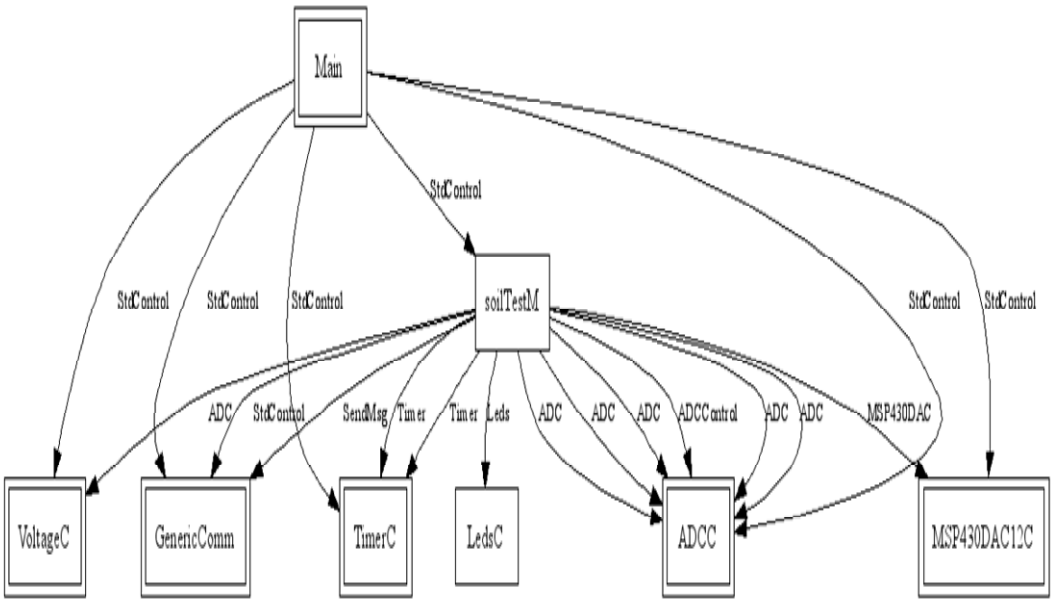


**Figure 9: Soil Moisture Sensing node Components Wiring**

***SoilTestM.nc***

```
/*
Program for Sensing node
Soil sensors attached:4


This program reads soil sensors readings(4) every 2 hours and stores in the RAM and
after reading for a period of one day(12 readings)then the data are sent to the central
node

This program uses DAC1.

DAC1 provides the required excitation(2.5V for the soil moisture sensor) on ports
ADC0,ADC1, ADC2 & ADC3.


Total bytes = (12 readings*2 bytes * 4 sensors)+10 bytes header+2 bytes node id +2
bytes voltage
                        =96+10+2+2
                        =110 bytes/day/node

maximum bytes that can be sent in 1 message is 128 bytes


Therefore the Packet structure(single packet) is as follows

        uint16_t adc_zero[6];   2bytes*12=24 bytes

        uint16_t adc_one[6];   2bytes*12=24 bytes

        uint16_t adc_two[6];   2bytes*12=24 bytes

        uint16_t adc_three[6];  2bytes*12=24 bytes


        uint16_t volt;                  2 bytes
```

**SoilTestM.nc**

```
        uint16_t node_id;       2 bytes
        ----------------------------------------
        Total = 100 bytes+10 bytes header=110 bytes




*/

includes MyMsg;


module soilTestM
{
  provides interface StdControl;
  uses {
    interface Timer as Timer1;    // Timer 1 repeats every two hrs
        interface Timer as Timer2;      // on DAC1 for timeinterval of 10MS (required a
minimum 10ms after the excitation ond the DAC)

    interface Leds;
    interface StdControl as CommControl;

        interface SendMsg as Send;

        interface ADCControl;

        interface ADC as Soilsensor;  //senosr1
        interface ADC as Soilsensor1;  //senosr2
        interface ADC as Soilsensor2; //senosr3
        interface ADC as Soilsensor3; //senosr4

        interface ADC as Soilsensor6;

        //interface MSP430DAC as DAC0;
        interface MSP430DAC as DAC1;


        interface ADC as Voltage;
```

```
  }
}

implementation
{
TOS_Msg pkt;
bool busy;

 // To store soil sensor values locally before sending
uint16_t soil[12];
uint16_t soil1[12];
uint16_t soil2[12];
uint16_t soil3[12];

uint16_t i=0,k=0,count=1;
uint16_t j=0,level_flag;
uint16_t voltage_read;
bool flag=TRUE;
task void temp();
//task void temp1();
task void sendData();

  /**
   * Used to initialize this component.
   */

  command result_t StdControl.init() {

          call Leds.init();

    call ADCControl.init();


//Binding ADC0,ADC1,ADC2,ADC3 ports, ports are specified in MyMsg.h header file
where INPUT_CHANNEL refers to the ADC port
```

```
   call ADCControl.bindPort( TOS_ADC_Soilsensor_PORT,
TOSH_ACTUAL_ADC_Soilsensor_PORT );

   call ADCControl.bindPort( TOS_ADC_Soilsensor1_PORT,
TOSH_ACTUAL_ADC_Soilsensor1_PORT );

   call ADCControl.bindPort( TOS_ADC_Soilsensor2_PORT,
TOSH_ACTUAL_ADC_Soilsensor2_PORT );

   call ADCControl.bindPort( TOS_ADC_Soilsensor3_PORT,
TOSH_ACTUAL_ADC_Soilsensor3_PORT );


   call ADCControl.bindPort( TOS_ADC_Soilsensor6_PORT,
TOSH_ACTUAL_ADC_Soilsensor6_PORT );


//Binding the DAC



   call DAC1.bind(DAC12_REF_VREF, //internal voltage reference
//²úÉúµçÑ¹µÄÔ¼Êø£¬Ïàµ±ÓÚ¿ØÖÆ¼Ä´æÆ÷Éè¶¨
          DAC12_RES_12BIT, //resolution 12 bit
          DAC12_LOAD_WRITE, //write directly
          DAC12_FSOUT_1X, //1X voltage
          DAC12_AMP_MED_HIGH, //medium response time & power
          DAC12_DF_STRAIGHT,//unsigned int
          DAC12_GROUP_OFF);//group off, using only one DAC


   return SUCCESS;
  }


  /**
   * Starts the SensorControl component.
   * @return Always returns SUCCESS.
   */
  command result_t StdControl.start() {
```

```
  //call Timer1.start( TIMER_REPEAT,300000); // to read sensor values every 2 hr
    call Timer1.start( TIMER_REPEAT,7200000); // to read sensor values every 2 hr
          //call Timer1.start( TIMER_REPEAT,2000); // to read sensor values every 2 hr
    return SUCCESS;
  }



  /**
   * Stops the SensorControl component.
   * @return Always returns SUCCESS.
   */
  command result_t StdControl.stop() {

    call Timer1.stop();
          call Timer2.stop();
          //call Timer3.stop();
    return SUCCESS;
  }



//Enable the DAC0

  event result_t Timer1.fired() {

          call DAC1.enable();

    return SUCCESS;
  }



  //Read sensor values on ADC0 and ADC1

   event result_t Timer2.fired() {
```

```
  call Leds.yellowOn();
  call Soilsensor.getData();

  return SUCCESS;
 }




//Read soil moisture sensor value on port adc0 when data is ready

 async event result_t Soilsensor.dataReady( uint16_t data ) {

   soil[i]=data;
        call Soilsensor1.getData();
        return SUCCESS;

}

//Read soil moisture sensor value on port adc1 when data is ready
 async event result_t Soilsensor1.dataReady( uint16_t data ) {

        soil1[i]=data;
        //call Leds.redOn();
   call Soilsensor2.getData();
        return SUCCESS;
}

//Read soil moisture sensor value on port adc2 when data is ready
 async event result_t Soilsensor2.dataReady( uint16_t data ) {

   soil2[i]=data;
        call Soilsensor3.getData();
        return SUCCESS;

}
```

**SoilTestM.nc**

```
//Read soil moisture sensor value on port adc3 when data is ready
 async event result_t Soilsensor3.dataReady( uint16_t data ) {

        soil3[i]=data;
        //call Leds.redOn();
   //post temp1();
        post temp();
   i++;
   return SUCCESS;


}


//get the voltage from ADC6 channel
async event result_t Soilsensor6.dataReady( uint16_t data ) {

        voltage_read=data;
        post sendData();

}


event result_t Send.sendDone(TOS_MsgPtr msg, result_t success)
  {
        //call Leds.greenOn();

        return SUCCESS;

  }

event void DAC1.enableDone(result_t success)
        {
           call DAC1.enableOutput();
                //call DAC0.set(4060);  // set it 2.5V
                call DAC1.set(4070);  // set it 2.5V  For 2.4
                call Timer2.start( TIMER_ONE_SHOT, 10); //time interval of 10ms
        }
```

**SoilTestM.nc**

```
/*event void DAC1.enableDone(result_t success)
        {
      call DAC1.enableOutput();
                call DAC1.set(4080);  // set it 2.5V
                call Timer3.start( TIMER_ONE_SHOT, 60000); //time interval of 10ms
        }*/


event void DAC1.disableDone(result_t success){

           // Check to see a packet has to be sent to the central node
                if(i==12)
           {
           i=0;
                // call Voltage.getData();
                 call Soilsensor6.getData();
           }

        }

/*event void DAC1.disableDone(result_t success){




        }*/


        task void temp()
        {

                //Disabling the DAC0
                 call DAC1.set(0);
                 call DAC1.disableOutput();
                 call DAC1.disable();


        }
```

```
        /*task void temp1()
        {

                //Disabling the DAC1
                call DAC1.set(0);
                call DAC1.disableOutput();
                call DAC1.disable();


        }*/



        async event result_t Voltage.dataReady(uint16_t data) {

        voltage_read=data;

// Next line must be commented if data has to be sent on button pressed.
        post sendData();

   return SUCCESS;
  }

  //Send the sensor values the central node

  task void sendData()
  {


        MyMsg *MyData = (MyMsg *)pkt.data;
        call Leds.greenToggle();



        MyData->volt=voltage_read;
        MyData->node_id=TOS_LOCAL_ADDRESS;
        k=0;

        //copy values to the data packet
```

**SoilTestM.nc**

```
        for(j=0;j<12;j++)
        {
                MyData->adc_zero[j]=soil[j];
                MyData->adc_one[j]=soil1[j];
                MyData->adc_two[j]=soil2[j];
                MyData->adc_three[j]=soil3[j];

        }

        //send the data packet
                if((call Send.send(0, sizeof(MyMsg), &pkt)) == SUCCESS)
                {

                }
        }


}
```

**SoilTest.nc**

```
//Config file
includes MyMsg;


configuration soilTest { }   // A configuration be used by other configurations
implementation
{
  components Main           // A high-level configuration? It provides Main here.
        , soilTestM, TimerC, LedsC,  GenericComm as Comm,ADCC,
MSP430DAC12C,VoltageC;


  Main.StdControl -> TimerC;   //for timers
  Main.StdControl -> Comm;    //for sending and receiving messages
  Main.StdControl -> MSP430DAC12C; //for DAC
  Main.StdControl -> VoltageC;   //for voltage
  Main.StdControl -> soilTestM;
  Main.StdControl -> ADCC;   //for ADC channels
```

***SoilTest.nc***

```
  soilTestM.Timer1 -> TimerC.Timer[unique("Timer")];
  soilTestM.Timer2 -> TimerC.Timer[unique("Timer")];


  soilTestM.Send -> Comm.SendMsg[AM_STEST];
  soilTestM.CommControl -> Comm;


  soilTestM.Leds -> LedsC;

//binding to the ADC channels ..please check the MyMsg.h for channels

  soilTestM.Soilsensor -> ADCC.ADC[TOS_ADC_Soilsensor_PORT];
  soilTestM.Soilsensor1 -> ADCC.ADC[TOS_ADC_Soilsensor1_PORT];
  soilTestM.Soilsensor2 -> ADCC.ADC[TOS_ADC_Soilsensor2_PORT];
  soilTestM.Soilsensor3 -> ADCC.ADC[TOS_ADC_Soilsensor3_PORT];

  soilTestM.Soilsensor6 -> ADCC.ADC[TOS_ADC_Soilsensor6_PORT];
  soilTestM.ADCControl -> ADCC;

  //soilTestM.DAC0 -> MSP430DAC12C.DAC0;
  soilTestM.DAC1 -> MSP430DAC12C.DAC1;

  soilTestM.Voltage -> VoltageC;


}
```

***Header Files***

*MyMsg.h*

```
typedef struct MyMsg{

        uint16_t adc_zero[12]; //for storing soil moisture sensor1 reading for 12
readings
        uint16_t adc_one[12]; //for storing soil moisture sensor2 reading for 12 readings
        uint16_t adc_two[12];//for storing soil moisture sensor3 reading for 12 readings
```

*MyMsg.h*

```
        uint16_t adc_three[12];//for storing soil moisture sensor4 reading for 12
readings


        uint16_t volt; //for storing voltage readings
        uint16_t node_id; ////for storing node id
}MyMsg;

enum {
        AM_STEST = 55  //Message id
};


//for connection to channel ADC0 for sensor1
enum {

  TOS_ADC_Soilsensor_PORT = unique("ADCPort"),

  TOSH_ACTUAL_ADC_Soilsensor_PORT = ASSOCIATE_ADC_CHANNEL(
    INPUT_CHANNEL_A0,  //A0 represents ADC0
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5  //2_5 represents voltage 2.5
  ),
};

//for connection to channel ADC1 for sensor2
enum {

  TOS_ADC_Soilsensor1_PORT = unique("ADCPort"),

  TOSH_ACTUAL_ADC_Soilsensor1_PORT = ASSOCIATE_ADC_CHANNEL(
    INPUT_CHANNEL_A1, //A1 represents ADC1
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5 //2_5 represents voltage 2.5
  ),
};

//for connection to channel ADC2 for sensor3
enum {
```

*MyMsg.h*

```
  TOS_ADC_Soilsensor2_PORT = unique("ADCPort"),    // What does unique mean?

  TOSH_ACTUAL_ADC_Soilsensor2_PORT = ASSOCIATE_ADC_CHANNEL(
    INPUT_CHANNEL_A2, //A2 represents ADC2
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5 //2_5 represents voltage 2.5
  ),
};

//for connection to channel ADC6 for reading voltage
enum {

  TOS_ADC_Soilsensor6_PORT = unique("ADCPort"),    // What does unique mean?

  TOSH_ACTUAL_ADC_Soilsensor6_PORT = ASSOCIATE_ADC_CHANNEL(
    INPUT_CHANNEL_A6, //A6 represents ADC6
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5 //2_5 represents voltage 2.5
  ),
};
//for connection to channel ADC3 for sensor4
enum {

  TOS_ADC_Soilsensor3_PORT = unique("ADCPort"),    // What does unique mean?

  TOSH_ACTUAL_ADC_Soilsensor3_PORT = ASSOCIATE_ADC_CHANNEL(
    INPUT_CHANNEL_A3, //A3 represents ADC3
    REFERENCE_VREFplus_AVss,
    REFVOLT_LEVEL_2_5 //2_5 represents voltage 2.5
  ),
};
```

Base Node

*List of files:*

- *BasenodeM.nc*
- *Basenode.nc*
- *receiveMsg.h*
- *MyMsg*
- *Modfied Listen.java file can be found under \cygwin\opt\tinyos-1.x\tools\java\net\tinyos\tools\Listen.java*
- *\cygwin\opt\moteiv\apps\TosBase*

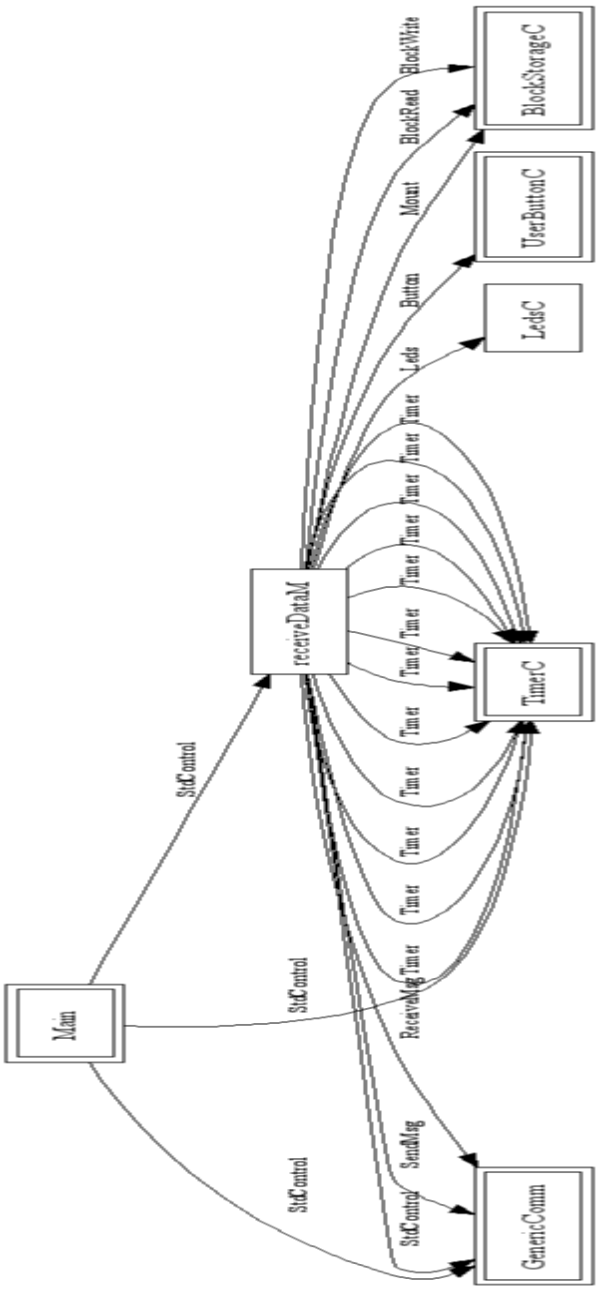


**Figure 10: Soil Moisture Base Node Components Wiring**

***BaseNode.nc***

```
includes MyMsg;
includes receiveMsg;

configuration receiveData{ }

implementation
{

  components Main
          , receiveDataM,
            TimerC,
            LedsC,
            BlockStorageC,
            GenericComm as Comm,
           UserButtonC;



enum { ID = unique("Blockstorage") };

  Main.StdControl -> TimerC;
  Main.StdControl -> Comm;

  Main.StdControl -> receiveDataM;

  receiveDataM.Leds -> LedsC;
  receiveDataM.Button -> UserButtonC; //for button pressed event

  receiveDataM.Timer1 -> TimerC.Timer[unique("Timer")];
  receiveDataM.Timer2 -> TimerC.Timer[unique("Timer")];

  receiveDataM.Timer3 -> TimerC.Timer[unique("Timer")];
  receiveDataM.Timer4 -> TimerC.Timer[unique("Timer")];

receiveDataM.Timer5 -> TimerC.Timer[unique("Timer")];
```

**BaseNode.nc**

```
  receiveDataM.Timer6 -> TimerC.Timer[unique("Timer")];
  receiveDataM.Timer7 -> TimerC.Timer[unique("Timer")];
  receiveDataM.Timer8 -> TimerC.Timer[unique("Timer")];

  receiveDataM.Timer9 -> TimerC.Timer[unique("Timer")];
  receiveDataM.Timer10 -> TimerC.Timer[unique("Timer")];
  receiveDataM.Timer11 -> TimerC.Timer[unique("Timer")];

  receiveDataM.Timer12 -> TimerC.Timer[unique("Timer")];



  receiveDataM.CommControl -> Comm;  //for sending and receiving messages
  receiveDataM.Send -> Comm.SendMsg[AM_RTEST]; //for send
  receiveDataM.Receive -> Comm.ReceiveMsg[AM_STEST]; //for receive

  receiveDataM.Mount -> BlockStorageC.Mount[ID];
  receiveDataM.BlockRead -> BlockStorageC.BlockRead[ID];
  receiveDataM.BlockWrite -> BlockStorageC.BlockWrite[ID];
}
```

**BaseNodeM.nc**

```
/* This program acts as a central node .

Receive event is the Main event in the program

Whenever a message is received by this node the data is stored on the node's flash based
on the node id
since there can be 256 pages in one block, on each page the data for a particular node is
stoted,

Flash consists of 16 blocks
each block has  256 pages
each page has 256 bytes

A particular node's data is stored in one single page (node_id*256)
```

**BaseNodeM.nc**

```
*/

includes MyMsg;
includes receiveMsg;

module receiveDataM
{
  provides interface StdControl;
  uses {
    interface Timer as Timer1;
        interface Timer as Timer2;

        interface Timer as Timer3; //node2
        interface Timer as Timer4; //node3
        interface Timer as Timer5; //node4

        interface Timer as Timer6; //node5
        interface Timer as Timer7; //node6
        interface Timer as Timer8; //node7


        interface Timer as Timer9; //node8
        interface Timer as Timer10; //node9
        interface Timer as Timer11; //node10

        interface Timer as Timer12; //node11

   interface Leds;
    interface StdControl as CommControl;
        interface SendMsg as Send;

        interface Mount;
        interface BlockRead;
        interface BlockWrite;
        interface ReceiveMsg as Receive;


         interface Button;

  }
```

```
}

implementation
{

        TOS_Msg pkt;

        uint16_t t[60];

        uint16_t w_buffer[60];  //write buffer
        uint16_t r_buffer[60]; //read buffer


//counters for all nodes

        uint16_t node2_cnt=0,count_node2=0;
        uint16_t node3_cnt=0,count_node3=0;

        uint16_t node4_cnt=0,count_node4=0;
        uint16_t node5_cnt=0,count_node5=0;

        uint16_t node6_cnt=0,count_node6=0;
        uint16_t node7_cnt=0,count_node7=0;

        uint16_t node8_cnt=0,count_node8=0;
        uint16_t node9_cnt=0,count_node9=0;

        uint16_t node10_cnt=0,count_node10=0;
        uint16_t node11_cnt=0,count_node11=0;


        uint16_t sending_node_id;
        uint16_t j=0,k=0,i=0;
        bool busy;
        bool ok;


 /*This function writes the data to the flash based on the node's id.
 Based on (nodeid's staring address+(packet count*256))
 nodeid's starting address refers to 0 for node 2,
```

```
        for node id 3 starting address is 5376 refers to page 22 (i.e. 256*21) since 21
pages is used for node id 2
 BlockWrite.write is the command used to write data blocks to the flash

  call BlockWrite.write(0+(node2_cnt*256),&w_buffer,100);
  0+(node2_cnt*256) refers to the address where the data has to be written on the flash
  &w_buffer refers to the data to be written
  100 refers to the number of bytes to be written (100 bytes refers to the data packet sent
by each node
 */

        task void write_to_flash(){
        /*MyMsg* MyData =(MyMsg *)pkt.data;
for(j=0;j<50;j++)
        {


    MyData->val[j]=w_buffer[j];
   }
        call Send.send(0, sizeof(MyMsg), &pkt);*/

        if(sending_node_id==2)
        {

                call BlockWrite.write(0+(node2_cnt*256),&w_buffer,100);
                node2_cnt++;
        }

        if(sending_node_id==3)
        {

                call BlockWrite.write(5376+(node3_cnt*256),&w_buffer,100);
                node3_cnt++;
        }

        if(sending_node_id==4)
        {

                call BlockWrite.write(10752+(node4_cnt*256),&w_buffer,100);
                node4_cnt++;
```

```
}

if(sending_node_id==5)
{

        call BlockWrite.write(16128+(node5_cnt*256),&w_buffer,100);
        node5_cnt++;
}

if(sending_node_id==6)
{

        call BlockWrite.write(21504+(node6_cnt*256),&w_buffer,100);
        node6_cnt++;
}

if(sending_node_id==7)
{

        call BlockWrite.write(26880+(node7_cnt*256),&w_buffer,100);
        node7_cnt++;
}

if(sending_node_id==8)
{

        call BlockWrite.write(32256+(node8_cnt*256),&w_buffer,100);
        node8_cnt++;
}

if(sending_node_id==9)
{

        call BlockWrite.write(37632+(node9_cnt*256),&w_buffer,100);
        node9_cnt++;
}

if(sending_node_id==10)
{
```

```
                call BlockWrite.write(43008+(node10_cnt*256),&w_buffer,100);
                node10_cnt++;
        }

        if(sending_node_id==11)
        {

                call BlockWrite.write(48384+(node11_cnt*256),&w_buffer,100);
                node11_cnt++;
        }
                 //100 bytes 96 bytes data soil+2 bytes node id+2 bytes voltage

   }


                /* This function is used for reading data from each node

                */
        task void reading(){


                call Timer3.start( TIMER_ONE_SHOT,2000);  //node2 : Reading node2
from flash 21 pages
                call Timer4.start( TIMER_ONE_SHOT,4000);  //node3
                call Timer5.start( TIMER_ONE_SHOT,6000);  //node4

                call Timer6.start( TIMER_ONE_SHOT,8000);  //node5
                call Timer7.start( TIMER_ONE_SHOT,10000);  //node6
                call Timer8.start( TIMER_ONE_SHOT,12000);  //node7

                call Timer9.start( TIMER_ONE_SHOT,14000);  //node8
                call Timer10.start( TIMER_ONE_SHOT,16000);  //node9
                call Timer11.start( TIMER_ONE_SHOT,18000);  //node10

                call Timer12.start( TIMER_ONE_SHOT,20000);  //node11

        }

         /* This function is used for retrieving data from flash and sending the data back
to the base station
```

```
BlockRead.read command is used to read the data from the flash
call BlockRead.read(0+(count_node2*256),&r_buffer,100);

0+(count_node2*256) refers to the starting address of node id 2
&r_buffer copy the contents to this address
100 refers to the number of bytes to be read from the particular address

*/

task void read_flash_node2()
{

        if(count_node2<21)
 {

 call BlockRead.read(0+(count_node2*256),&r_buffer,100);
 count_node2++;

        }
}

task void read_flash_node3()
{

        if(count_node3<21)
 {

 call BlockRead.read(5376+(count_node3*256),&r_buffer,100);
 count_node3++;

        }
}

 task void read_flash_node4()
{

        if(count_node4<21)
 {

 call BlockRead.read(10752+(count_node4*256),&r_buffer,100);
```

```
 count_node4++;

        }
}

task void read_flash_node5()
{
call Leds.redToggle();

        if(count_node5<21)
 {

 call BlockRead.read(16128+(count_node5*256),&r_buffer,100);
 count_node5++;

        }
}


task void read_flash_node6()
{

        if(count_node6<21)
 {

 call BlockRead.read(21504+(count_node6*256),&r_buffer,100);
 count_node6++;

        }
}

task void read_flash_node7()
{

        if(count_node7<21)
 {

 call BlockRead.read(26880+(count_node7*256),&r_buffer,100);
 count_node7++;
```

```
        }
}


task void read_flash_node8()
{

        if(count_node8<21)
 {

 call BlockRead.read(32256+(count_node8*256),&r_buffer,100);
 count_node8++;

        }
}

task void read_flash_node9()
{

        if(count_node9<21)
 {

 call BlockRead.read(37632+(count_node9*256),&r_buffer,100);
 count_node9++;

        }
}

task void read_flash_node10()
{

        if(count_node10<21)
 {

 call BlockRead.read(43008+(count_node10*256),&r_buffer,100);
 count_node10++;

        }
}
```

```
        task void read_flash_node11()
        {

                if(count_node11<21)
         {

         call BlockRead.read(48384+(count_node11*256),&r_buffer,100);
         count_node11++;

                }
        }

command result_t StdControl.init() {

        call Leds.init();
        call Mount.mount(1);

        //call BlockWrite.erase();
  return SUCCESS;
}


/**
 * Starts the SensorControl component.
 * @return Always returns SUCCESS.
 */
command result_t StdControl.start() {

  call Button.enable();
  return SUCCESS;

}




/**
 * Stops the SensorControl component.
 * @return Always returns SUCCESS.
 */
```

```
  command result_t StdControl.stop() {
    call Timer1.stop();
        call Timer2.stop();

        call Timer3.stop();
        call Timer4.stop();

        call Timer5.stop();
        call Timer6.stop();

        call Timer7.stop();
        call Timer8.stop();

        call Timer9.stop();
        call Timer10.stop();

        call Timer11.stop();
        call Timer12.stop();
    return SUCCESS;
  }


/*****************************************************/
/*          Timer1 & Timer2 do nothing here          */
/*****************************************************/
  event result_t Timer1.fired() {

    return SUCCESS;

  }

  event result_t Timer2.fired() {

    return SUCCESS;

  }


  event result_t Timer3.fired() {
```

```
        post read_flash_node2();
  return SUCCESS;

  }

  event result_t Timer4.fired() {

  post read_flash_node3();
    return SUCCESS;

  }

event result_t Timer5.fired() {

  post read_flash_node4();
    return SUCCESS;

  }


event result_t Timer6.fired() {

  post read_flash_node5();
    return SUCCESS;

  }

event result_t Timer7.fired() {

  post read_flash_node6();
    return SUCCESS;

  }

event result_t Timer8.fired() {

  post read_flash_node7();
    return SUCCESS;
```

```
  }

event result_t Timer9.fired() {

  post read_flash_node8();
    return SUCCESS;

  }

event result_t Timer10.fired() {

  post read_flash_node9();
    return SUCCESS;

  }

event result_t Timer11.fired() {

  post read_flash_node10();
    return SUCCESS;

  }

event result_t Timer12.fired() {

  post read_flash_node11();
    return SUCCESS;

  }



/*****************************************************/


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


        return SUCCESS;
```

```
 }

event void Mount.mountDone(storage_result_t result, volume_id_t id) {
        /*if (result==SUCCESS)
          {call Leds.greenOn();}
   else
          {call Leds.redOn(); }*/
 }



event void BlockWrite.eraseDone(storage_result_t result) {

 }

event void BlockWrite.commitDone(storage_result_t result) {


 }

event void BlockWrite.writeDone(storage_result_t result, block_addr_t addr, void *buf,
block_addr_t len) {

 call BlockWrite.commit();
 call Leds.redOn();

 }

event void BlockRead.computeCrcDone(storage_result_t result, uint16_t crc,
block_addr_t addr, block_addr_t len) {

 }

event void BlockRead.readDone(storage_result_t result, block_addr_t addr, void *buf,
block_addr_t len) {

MyMsg* MyData =(MyMsg *)pkt.data;
call Leds.yellowToggle();

          for(j=0;j<50;j++)
```

**BaseNodeM.nc**

```
                {
                 MyData->val[j]=r_buffer[j];
                }
          call Send.send(0, sizeof(MyMsg), &pkt);

          call Timer3.start( TIMER_ONE_SHOT,2000); //node2
          call Timer4.start( TIMER_ONE_SHOT,3000); //node3
          call Timer5.start( TIMER_ONE_SHOT,4000); //node4

          call Timer6.start( TIMER_ONE_SHOT,5000); //node5
          call Timer7.start( TIMER_ONE_SHOT,6000); //node6
          call Timer8.start( TIMER_ONE_SHOT,7000); //node7

          call Timer9.start( TIMER_ONE_SHOT,8000); //node8
          call Timer10.start( TIMER_ONE_SHOT,9000); //node9
          call Timer11.start( TIMER_ONE_SHOT,10000); //node10

          call Timer12.start( TIMER_ONE_SHOT,11000);//node11

 }

event void BlockRead.verifyDone(storage_result_t result) {

}



//when a message is received

          event TOS_MsgPtr Receive.receive(TOS_MsgPtr m) {


          //MyMsg* MyData =(MyMsg *)m->data;

          receiveMsg* receiveData =(receiveMsg *)m->data;
          call Leds.greenOn();

                    sending_node_id=m->addr;

          for(i=0;i<50;i++)
```

```
{
        w_buffer[i]=0;
}

i=0;
for(j=0;j<12;j++,i++)
{
        w_buffer[i]=receiveData->adc_zero[j];

}
i=12;
for(j=0;j<12;j++,i++)
{
        w_buffer[i]=receiveData->adc_one[j];

}
i=24;
for(j=0;j<12;j++,i++)
{
        w_buffer[i]=receiveData->adc_two[j];

}
i=36;
for(j=0;j<12;j++,i++)
{
        w_buffer[i]=receiveData->adc_three[j];

}


        w_buffer[48]=receiveData->volt;

        w_buffer[49]=receiveData->node_id;


sending_node_id=w_buffer[49];

//call Send.send(TOS_UART_ADDR,sizeof(receiveMsg),&pkt);
//call Mount.mount(1);
post write_to_flash();
```

**BaseNodeM.nc**

```
        return m;
  }

async event void Button.pressed( uint32_t when ) {


   post reading();

  }

async event void Button.released( uint32_t when ) {

  }

}
```

***Header Files***

**MyMsg.h**

***Header File***

```
typedef struct MyMsg{


        uint16_t val[50];
}MyMsg;

enum {
        AM_RTEST = 35
};
```

**ReceiveMsg.h**

```
// Receiveing message from sensing node data packet format same as sending node's
typedef struct receiveMsg{
uint16_t adc_zero[12];
        uint16_t adc_one[12];
        uint16_t adc_two[12];
```

**ReceiveMsg.h**

```
        uint16_t adc_three[12];


        uint16_t volt;
        uint16_t node_id;

}receiveMsg;

enum {
        AM_STEST = 55
};
```

***Listen.java***

*// $Id: Listen.java,v 1.5 2004/08/19 00:13:49 idgay Exp $*
***Modified Listen.java file***

```
/*                                                                  tab:4
 * "Copyright (c) 2000-2003 The Regents of the University  of California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and its
 * documentation for any purpose, without fee, and without written agreement is
 * hereby granted, provided that the above copyright notice, the following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY
PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE
UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY
WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED
HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO
OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-LICENSE
 * file. If you do not find these files, copies can be found by writing to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley, CA,
 * 94704.  Attention:  Intel License Inquiry.
 */
```

***Listen.java***

```
package net.tinyos.tools;

import java.io.*;
import java.util.*;
import java.lang.Math;
import net.tinyos.packet.*;
import net.tinyos.util.*;
import net.tinyos.message.*;

public class Listen {
   public static void main(String args[]) throws IOException {

          int i;
      int cnt = 0;
      int flag=0;
                  Date yy = new Date();


         if (args.length > 0) {
            System.err.println("usage: java net.tinyos.tools.Listen");
            System.exit(2);
         }

         PacketSource reader = BuildSource.makePacketSource();
         if (reader == null) {
            System.err.println("Invalid packet source (check your MOTECOM environment
variable)");
            System.exit(2);
         }

         try {
           reader.open(PrintStreamMessenger.err);
           for (;;) {
             byte[] packet = reader.readPacket();
                   System.out.println("Packet Length"+packet.length);
```

**Listen.java**

```
                yy = new Date();


        int len = packet.length;
        System.out.print("packet length=" + len);
                System.out.print("NODE ID=" + packet[108]);

                //String xx = "C:/cygwin/opt/moteiv/apps/AgTest/" + "mote" +
packet[108]+".txt";

        String xx =
"C:/cygwin/opt/moteiv/SoilTestReadings/"+"Mote"+packet[108]+".txt" ;
        //sdf.setTimeZone(TimeZone.getDefault());
        File file = new File(xx);
        if (!file.exists())
        {
         //   System.out.println(xx+"\n");

          flag=1;

        }

        else
        {
        }
        FileWriter fstream = new FileWriter(xx, true);
        BufferedWriter out = new BufferedWriter(fstream);

        if (flag == 1)
        {
          // System.out.println("File Does not Exist");
           out.write("Sensor1"+"\t\t\t"+"Sensor2"  +"\t\t\t"+ "Sensor3" + "\t\t\t" +
"Sensor4"+"\t\t\t Time Packet Received");
           out.newLine();
           out.write("----------------------------------------------------------------------------------
--");
           out.newLine();
          flag = 0;
        }
        else
```

```
{
   //System.out.println("File Exist");
}



cnt++;
int[] temp = new int[packet.length];
for (i = 0; i < packet.length; i++)
{
   if (packet[i] < 0)
      temp[i] = packet[i] + 256;
   else
      temp[i] = packet[i];
}

            System.out.println("i value ="+i);
int kk = 0;
int j = 0, k = 0, m = 0, n = 0, o = 0,l=0;

double[] soil1=new double[12];
double[] soil2 = new double[12];
double[] soil3 = new double[12];
double[] soil4 = new double[12];

k=0;

    m=0;

for (i = 10; i < packet.length; )
{



   kk = 0;

   kk = kk | temp[i + 1];

   kk = kk << 8;
```

```
        kk = kk | temp[i];


       if(j<12)
                        {
                        soil1[j]=(double)((double)kk/4095)*2.5;
                        //System.out.println("Soil1 ="+soil1[j]+":::"+j);

                        }

                        else if(j>=12 && j<24)
                        {
                        soil2[k]=(double)((double)kk/4095)*2.5;
                       //System.out.println("Soil2 ="+soil2[k]+":::"+k);
                        k++;
                        }


                        else if(j>=24 && j<36)
                        {
                        soil3[l]=(double)((double)kk/4095)*2.5;
                       //System.out.println("Soil3 ="+soil3[l]+":::"+l);
                        l++;
                        }

                        else if(j>35 && j<48)
                        {
                        soil4[m]=(double)((double)kk/4095)*2.5;
                       //System.out.println("Soil4 ="+soil4[m]+":::"+m);
                        m++;
                        }


        j++;
         i = i + 2;

      }

              //  System.out.println("j value ="+j+" "+i);
       // Printing in the file
```

**Listen.java**

```
        for (i = 0; i < 12; i++)
        {
                    // System.out.println("i value ="+i);
                     j=0;
                     while(j<4)
                     {

                                 if(j==3)
                                 {

                                  break;
                                 }
                                 else
                                 {
                                //  System.out.println("j value ="+j);
                                  out.write((float)soil1[i]
+"\t\t"+(float)soil2[i]+"\t\t"+(float)soil3[i] +"\t\t"+(float)soil4[i] +"\t\t"+yy.toString());

 //out.write( Math.round(temperature[i] * 100.0) / 100.0 );
                                          out.newLine();

                                        }
                                         j++;
                                }


        }
                     if(i==12)
                     {
                     out.newLine();
                     out.newLine();
                     out.newLine();
                     out.newLine();
                     }

//new code ends

                Dump.printPacket(System.out, packet);
                System.out.println();
                System.out.flush();
```

**Listen.java**

```
                out.close();
            }
          }
          catch (IOException e) {
             System.err.println("Error on " + reader.getName() + ": " + e);
          }
    }
}
```

VITA

Arunkumar Venkateshawaran

Candidate for the Degree of

Master of Science or Arts

Thesis: DESIGN, IMPLEMENTAION AND EXPERIMENTS OF SENSING TASK SCHEDULING IN WIRELESS SENSOR NETWORKS

Major Field: Computer Science

Biographical:

Personal Data: Born in Coimbatore, India

Education:
Completed the requirements for the Master of Science in Computer Science at Oklahoma State University, Stillwater, Oklahoma in December, 2008.

Experience:

2006-2007 Graduate Assistant, Department of Bio Systems and Agriculture, Oklahoma State University.

2005-2006 Graduate Teaching Assistant, Computer Science Department, Oklahoma State University.

Name: Arunkumar Venkateshwaran Date of Degree: December, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: DESIGN, IMPLEMENTAION AND EXPERIMENTS OF SENSING TASK SCHEDULING IN WIRELESS SENSOR NETWORKS

Pages in Study: 138 Candidate for the Degree of Master of Science

Major Field: Computer Science

Divisible load scheduling theory (DLT) investigates the optimal distribution of partitionable loads among the processors and links in a system. Most of the recent work on DLT provides an efficient algorithm for task allocation to processors in a network by considering the processing time as well as the communication time. To minimize the execution time of a given task, this thesis proposes an efficient and integrated scheduling algorithm for sensing time and power management problem following the hierarchical divisible load scheduling paradigm. The proposed scheduling strategy minimizes the finish time by eliminating transmission collisions and idle gaps between two successive data transmissions. This thesis also addresses the power management issues by applying encoding techniques on the sensor data to improve the number of bytes sent in a single packet and thus improving the global power consumption of the network. Finally, this thesis also discusses how an application has been deployed in precision agriculture for effectively monitoring of the agriculture field and proposes an efficient way to increase the lifetime of the network by employing the encoding techniques.

ADVISER'S APPROVAL: Dr.Xiaolin Li