Technical Report: alg03-
August 23, 2003

Sunayana Saha, sunayana_ssaha@yahoo.com
Peter Bajcsy, pbajcsy@ncsa.uiuc.edu

Automated Learning Group
National Center for Supercomputing Applications
605 East Springfield Avenue, Champaign, IL 61820

System Design Issues for Applications Using Wireless Sensor Networks

Abstract
In this report we present a summary of the system design issues for applications using wireless "smart" MEMS sensor networks. We give an overview of facts we learnt from previous work done in this field, the challenges we encountered and the different experiments we performed during a system design using wireless sensor networks. This report presents our preliminary findings and describes the application that we are developing.

1. Introduction

Sensor networks are an emerging area of mobile computing. Networked sensors represent a new design paradigm enabled by advances in micro electro-mechanical systems (MEMS) and low power technology. Created with integrated circuit (IC) technology and combined with computational logic, these 'smart' sensors have the benefit of small size, low cost and power consumption, and, the capability to perform on-board computation. The built-in support for wireless communication makes it possible to deploy them in remote, humanly inaccessible locations. This recent technological innovation has opened doors for new ventures into an unthinkably large number of useful application domains. We will describe one such application below.

1.1 Problem statement and proposed solution

We have focused on an application of the MEMS sensors for camera calibration. There are many commercially available special cameras like the thermal infrared (IR) cameras, the hyperspectral cameras and visible and multi-spectral cameras. These cameras are useful in assessing the environment and analyzing it for hazards and other interesting or abnormal phenomena.

In this report we describe our experiences while calibrating the thermal IR camera. A thermal IR camera reacts to temperature changes in the scene it is shooting. It gives different pixel values for hot and cold objects in its field of view. Consider a thermal camera capturing the aerial view of an agricultural field. The temperature of different parts of the field could be different and this knowledge would be beneficial to an agriculturist. If we could map any pixel value from the image of the field taken by the thermal camera, into a temperature value, then we could determine the temperature of each and every pixel of the field. Another possible application would be using the thermal camera for indoor hazard monitoring. The advantage of deploying a thermal IR camera as opposed to a visible camera is that the visible camera would not sense spatially varying hazardous temperatures. The absence of light makes no difference in the operation of the thermal camera, which reacts mainly to temperature. Suppose the camera periodically captures images. These images are then displayed on a home-security inspector's computer screen. The inspector will study these images to determine normal conditions. But, how sufficient is the image alone to distinguish a normal situation from an abnormal one? A body may be warmer (and brighter) than other things around it, but it may still not require immediate human attention as opposed to fire flames or extremely hot objects, etc.

What would add great value to the utility of these cameras is a calibration for these cameras. Given a thermal camera image, how would one relate it to a particular temperature value? It is this calibration process that we are working on. More specifically, we want to develop a mapping that would allow us to usefully and reliably analyze the data from these cameras.

One might wonder where and how the MEMS sensors come into the picture. To find a mapping of pixel values to temperatures, we looked for a small cheap temperature measurement device that we could place in the environment and receive readings from it. The MEMS sensors appeared as a good choice for this purpose. MEMS sensor boards have different sensing devices attached to them. These sensors could sense ambient conditions like temperature, sound, movement, luminance etc. For calibrating the thermal camera we will use the thermistor or the temperature sensor. The thermistor reads the ambient temperature and outputs a raw reading using the analog-to-digital converter (ADC) attached to the sensor board. The raw reading is the voltage across the thermistor. The higher the temperature, the greater the voltage value. The raw readings can be converted to engineering units (degree Kelvin, for example) using a formula specified in the data sheet provided the manufacturers of the thermistor. We will present the formula in a later section.

The procedure for calibration of thermal IR images using MEMS sensor readings can be described as follows:

- Program MEMS sensors
- Place the sensors in a room and focus the thermal camera to capture images of the points where the sensors are placed.
- Synchronize MEMS sensors and thermal IR camera
- Collect wirelessly MEMS readings over time
- Collect thermal IR images over time
- Convert the raw ADC values from the MEMS sensors to engineering units (ºC) using the conversion formula (given in a later section).
- Identify MEMS sensor locations in a corresponding thermal IR image to obtain pairs o MEMS & thermal IR pixel values.
- Establish the mapping between the thermal IR pixel values and MEMS temperature values and calibrate the thermal IR image.

The above description sounds quite straightforward but it is only a bird's eye view of the actual work we had to do in order to get the desirable outcome. The report is structured as follows: We start by describing the MEMS sensors in Section 2. This includes laying out the specifications of the sensors we worked with, the on-going work in the research community for possible applications of these sensors, and, the issues we learnt both from the wireless sensor literature and our own experiences. We then move on to more specific topics that we had to handle for our application. Section 3 describes the issues in acquiring and analyzing data from the thermal camera. Section 4 presents the issues related to using the camera together with the sensors. Section 5 explains the experiments we conducted for calibrating the camera and the results obtained. Finally, we draw our conclusions in Section 6 and discuss the direction for future work.

2. MEMS sensors

This section describes the MEMS sensors. It gives the sensor specifications and some details of the software suite used to programme the sensors. It also enlists some prototype sensor network applications, followed by the problems faced by sensor applications. The proposed solutions to some of the problems can serve as guidelines for future application design. Finally, it discusses in detail the challenges specific to our application. In this report we will use the term sensor *'mote'* and sensor *'node'* interchangeably. 'Mote' is a commonly coined term in MEMS sensor literature.

2.1 Sensor specification

In order to understand the capabilities and limitations of MEMS sensors, we need to know the specifications of the sensors [1]. The sensors that we used for this study were bought from Crossbow Technology Inc. [2]. Our sensors were mounted on MTS101CA [3] sensor board (see Figure 1). The MTS101CA series sensor boards have a precision thermistor and a light/photocell sensor. A sensor board must be attached to a board, a MICA mote in our case (see Figure 2), which makes the computational power and wireless capabilities available to the sensors. The Mica board specifications are:

- Plug-in sensor boards like the MTS101CA, attached through a 51-pin expansion connector.
- TinyOS (TOS) operating system.
- 4MHz Atmega 128L processor.
- 128K bytes Flash, 4K bytes SRAM and 4K bytes of EEPROM.
- 916MHz radio transceiver with a maximum data rate of 40Kbits/sec.
- Attached AA (2) battery pack.

According to the data sheet for the thermistor on MTS101CA [3], the raw reading from the ADC can be converted to degrees Kelvin with accuracy of ±0.2 Kelvin and using the following approximation over 0-50 ºC:

1/T(K) = a + b*Ln(Rthr) + c*[Ln(Rthr)]^3

where:

Rthr = R1(ADC_FS-ADC)/ADC
a = 0.001010024
b = 0.000242127
c = 0.000000146
R1 = 10K
ADC_FS = 1024

ADC = output value from sensor's ADC measurement.

Temperature in º C = (Temperature value in Kelvin) – 273

Figure 1 shows a MICA board. Figure 2 is a block diagram of the board shown in Figure 1.

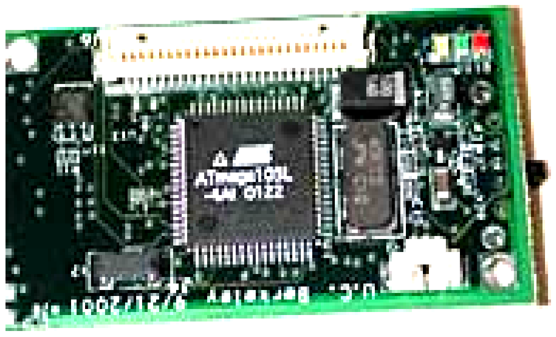

Figure 1 Mica Board for MTS101CA

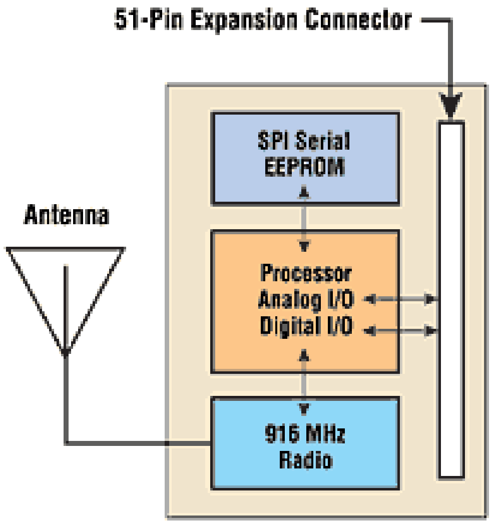

Figure 2 Block Diagram of Mica Mote

A Mica mote may cost (depending on the set of sensors attached to it), around 100 – 300 dollars and is about 4 inch by 2 inch by 1 inch in size. The computational, communication and sensing features together with the low cost and size make these MEMS sensors a very attractive alternative to their traditional counterparts. However, all the above capabilities have severe limitations. Sensors motes have limited processing and storage capability. Moreover, since AA batteries supply power to sensors, power becomes an expensive commodity. These limitations heavily influence any sensor application design and will be discussed in detail in the next section.

Sensor applications can be built by programming the sensors using a parallel port. MICA motes are programmed with TinyOS [4] code. TinyOS is an open source software platform developed by researchers at UC Berkeley and actively supported by a large community of users. It is a small operating system that allows networking, power management and sensor measurement details to be abstracted from core application development. TinyOS is optimized for efficient computational, energy and storage usage. The key to TinyOS's functionality is the NesC (network-embedded-systems-C) compiler, which is used to compile TinyOS programs. NesC has a C like structure and provides several advantages such as interfaces, wire error detection, automatic document generation, and facilitation of significant code optimizations. One of its major goals is to make TinyOS code cleaner, easier to understand, and easier to write.

TinyOS consists of a tiny scheduler and a graph of components [5]. A component has a set of *'command'* and *'event'* handlers. An event is like an interrupt sent from one component to another informing that something of interest to the receiver has occurred. For example, once an application's component has set the Timer component for a particular interval, the Timer component will inform the application through the 'Timer.fired()' *'event'*, when the specified interval is over. Conversely, a component can invoke another component's functionality using the callee component's *'command'* interface. An application would be using the command 'setTimer()' of the Timer component to use the timer functionality.

This clean abstraction allows rapid code development. Application development then involves plugging your components with the already implemented components - specifying how they will be linked to each other, that is, which component will invoke what command and which component will signal another about what events. This linkage is specified in a configuration file, which is one of the two files for any TinyOS component. The other file is the module file that contains the actual code for an application. Writing code is fairly simple if one has some basic knowledge in C programming.

TinyOS events are non-preemptive and are always run to completion. Thus, they must only do a small amount of work before completing otherwise they will stall the rest of the processing. In order to perform longer processing operations, TinyOS introduces a scheduling hierarchy consisting of *'tasks'* and *'events'*. Tasks are used to perform longer processing operations, such as background data processing and can be preempted by events. However, one task cannot preempt another task. Thus, a task should not spin or block for long periods of time. If one needs to run a series of long operations, one should have a separate task for each operation.

Keeping energy conservation in mind, the processor has three sleep modes: 'idle' which just shuts the processor off; 'power down', which shuts everything off except the watch-dog; and 'power save', which is similar to power-down, but leaves an asynchronous timer running. Power consumption equates to battery life. Long battery life is desired, and in some applications, one to five years is required. The processors, radio, and a typical sensor load consume a power of about 100 mW. This figure should be compared with the 30 µW drawn when all electrical components are inactive (in sleep mode). The overall system must embrace the philosophy of getting the work done as quickly as possible and then going into sleep mode.

While writing an application code, care must be taken in terms of power, CPU and memory usage. One should avoid writing too complex programs. The TinyOS architecture and scheduling policies must be kept in mind to write ‘efficient’ and ‘correct’ code. Careful planning and coding will lead to better application development and deployment. Appendix A presents a troubleshooting guide to some specific problems we faced during our work on the sensors.

2.2 Application domain

Sensor networks are revolutionizing the information-gathering process. Their low costs and small size make it possible to deploy them in large numbers, in inhospitable physical conditions such as remote geographical locations, toxic areas or less accessible places like machine interiors. The primary purpose of sensor networks is to sample the environment for sensory information and propagate this information wirelessly to bigger processing units or base station. There are many applications for wireless sensor networks [6] [7]. Some are new; others are traditional sensor applications that can be improved using wireless sensors. The overall list of applications includes:

- ❖ Physical security for military operations – for example, motion detection through accelerometer sensor for enemies’ army movement tracking in a battlefield.
- ❖ Environmental data collection – for example, habitat monitoring, bird/animal call recognition or tree temperature monitoring. Sensors are programmed with waveforms of bird and animal calls of interest. Sensors sense the environment and compare a bird/animal call with the one of interest. If there is a match, this information is passed wirelessly to the central processing unit.
- ❖ Seismic monitoring using a magnetometer.
- ❖ Industrial automation – sensors could keep track of the inventory levels and send signals to the base station if the inventory level goes below a specific level.
- ❖ Future consumer applications, including smart regulation of utilities at homes or detecting intruders. Sensors in such applications could help in detecting anomalous situations, or could simply be used to link different household devices. For instance, if we programme the sensors to detect high temperatures, they could wirelessly communicate with another device (say, a robot) to switch on the air conditioner. If sensors detect abnormally high temperatures, they could detect it as a fire and take action accordingly.

Though all these applications seem very appealing and useful, they are still in the early stages to development. A lot of barriers and challenges in wireless sensor networks have to be overcome in order to make the above applications successful. The next section points out of the difficulties encountered by the sensor network research community.

2.3 Challenges

In this section we will enumerate the challenges faced by any application in wireless sensor networks and discuss the solutions proposed in the sensor networks literature to work around these problems. The solutions vary from being wireless networking decisions to application design decisions. The proposed solutions form the foundation of any robust wireless sensor application. We also present the issues that are specific to our application. We should mention that the major challenges come from ad-hoc (multi-hop) networking and therefore we have focused first in our experimental work on single-hop networking operation scenarios.

2.3.1 Power constraints

Power remains to be the most valuable resource for a functional mote. Without power, everything, from communication, sensing to computation will come to a standstill. Even after careful coding and use of sleep modes when the sensors are idle, as discussed earlier, one must come to terms with the lifetime of a sensor mote, operating on a 3V power source, typically two AA batteries. Sooner or later the battery will run out of power, making the mote ineffective. The sensors will be deployed in remote areas, and so, rejuvenating them with new batteries might not always be an option. Since, losing a mote would mean loss of information from that part of the sensory field, it is important to introduce redundancy in the network. Even if one mote dies out, there should be others that still keep the network up and running. Redundancy is also crucial due to a mote's susceptibility to failure. It is hoped that one day, these motes would become so inexpensive, that discarding them, once they become inoperable due to power dissipation or failure, will be a cheaper alternative than repairing them. Along with redundancy, one needs a self-configuring network infrastructure. The network routing protocols should be such that the failure of one intermediate forwarding node should not deter other motes from communicating with the central station.

Power is consumed by the radio for receiving and transmitting messages. Transmission failures due to collisions in the network result in power wastage because the motes have to retransmit the same message until it is successfully delivered. Therefore, routing strategies are required and a network stack is desirable in order to minimize collisions and save power. There are numerous protocols to minimize energy wastage in message transmission. PAMAS [8] and S-MAC [9] are some commonly cited protocols that are variants of the MACAW protocol (used in traditional wireless networking domain). Traditional networking protocols require handshaking mechanisms between the sender and the receiver before the actual data transmission, e.g., RST (RequestToSend) and CTS (ClearToSend) signals. They also require motes to listen to the channel and only use it if the channel is free. Both these techniques are wasteful for sensor networks. Sensor network protocols suggest careful and limited use of handshaking, as transmissions are very energy intensive. They also suggest minimizing idle listening. A mote should put itself to sleep if it finds that the channel is busy, meaning the mote cannot anyway transmit.

Another term coined in the sensor network literature is *'data-aggregation'*. Instead of each mote sending its own packet, intermediate motes can aggregate (e.g. averaging all packet values or doing duplicate detection and suppression) packets from their neighbors thus reducing the number of transmissions flowing towards the final destination. Reduced transmissions mean decreased possibility of collisions, which in turn gives energy savings.

In short, we need robust and energy aware routing protocols and application designs. Applications should be able to tolerate some amount of failure. They can extend the lifetime of motes using techniques like data aggregation, at the cost of some delay introduced by the processing at the intermediate nodes.

2.3.2 Memory and CPU constraints

Another important issue to consider is memory and CPU constraints. Instead of allowing large amounts of data to be stored in sensor motes, applications should let motes forward small amounts of data towards the base station or the central processing unit. Some research papers suggest that we should avoid using the motes for complex computations. The motes should primarily collect information from the environment and do some minimal amount of calculations, like duplicate data detection or filtering of uninteresting events. A more powerful central processing station must handle computationally intensive tasks with the tradeoff of network transmission.

2.3.3 Application specific issues

While working with the MEMS sensors for our application, we have encountered a few interesting issues. Some of them were: (1) setting up the sensor network, (2) synchronizing the sensors, (3) time stamping the sensor readings, (4) data collection schemes, (5) MEMS sensor calibrations and (6) network evaluation approaches (a) by determining the rate of readings lost due to collisions during transmissions, and (b) by finding the extent to which factors like number of sensors, distance from base station, presence of other wireless devices, etc could affect data losses in our experimental setup. We tested different ways of collecting data from the sensors to find which one best suited our application. Each of these issues is addressed below.

2.3.3.1 Sensor network setup

The camera calibration application needs the sensors to be set up in an indoor environment. One of the sensors would be attached to a PC through a serial port. This sensor will be the base station, collecting data wirelessly from other sensors and forwarding it to the PC using the serial port. Other sensor motes would continuously sense the ambient temperature and record a temperature value after every 100 milliseconds. We chose to store a maximum of 10 readings on a mote because of our concern for limited memory on the motes. Also, owing to the fact that a larger packet has a greater chance of getting corrupted through collision, we preferred transmissions of smaller packet sizes. Since our camera calibration experiments were confined to a relatively small laboratory, we used a *'single hop'* sensor network to relay data from a sensor and the base station. Depending on the scheme used for data collection (discussed later), a sensor would send a packet containing 10 readings to the base station sensor. The base station will forward the data to the PC that will save this data to file for interpretation and analysis later.

2.3.3.2 Synchronizing the sensors

Synchronization of the sensors is a critical need of most applications and has several implications: (i) Starting the sensors at the same time allows us to correlate the readings of one sensor with another on a temporal domain. (ii) Instead of a sensor starting to sense the ambience the instant it is switched on, it is a better design if sensors wait for a signal to start. The base station could signal the sensors to start, when it is ready to receive data from the sensors. This will lead to huge energy savings, as it will eliminate the scenario where the sensors are sensing the environment and transmitting data when the base station is not even interested in the data. (iii) Synchronization is not just limited to synchronization among the sensors in the sensor network. A sensor network may be juxtaposed with other devices, for example, cameras. As with our application, we want to relate the thermal camera images to sensor readings, in which case, both the camera and the sensors must be synchronized together.

We handled the issue of synchronization by instructing the sensors to wait for a *'RESET'* signal from the base station. Until they receive this signal, they do not start sensing the environment. The base station broadcasts the *'RESET'* message. Once this message is received, a sensor starts sensing the ambience.

Theoretically, the precision to which the nodes are synchronized with each other depends on the possible sources of synchronization message latency – send time, access time, propagation time and receive time [10], [11]. If we only consider propagation time, then inter-node synchronization can be estimated as follows. Suppose the propagation delay of RESET message from the base station to a node is *'p'* time units. Then, if all sensors are placed at an equal distance from the base station, they will get the RESET message *'p'* time units after it is broadcast from the base station. Thus they will all be perfectly synchronized. If the sensors are placed at varying distances from the base station, they would get the RESET message between p_{min} and p_{max} time units, where p_{min} is the propagation delay to the closest sensor and p_{max} is the propagation delay to the farthest sensor. In this case, the nodes would be synchronized within a bound of '$p_{max} - p_{min}$' time units.

Another hurdle for synchronization is clock-drift. The hardware clocks on different nodes may count time at slightly differing rates. With progressing time, a particular sensor may be far ahead or behind some other sensor. This factor, together with the slight propagation delay of the synchronization message (as described above) can lead to a significant offset between different sensor timers, thus making them out of sync. A simple solution to rectify this asynchrony is by specifying an upper bound *'E'* for this asynchrony, and re-synchronizing the sensors at time *'t'*, after which the error goes beyond *'E'*. If we know the clock drift rate *'d'* (a clock manufacturer specified value), propagation delays p_{min} and p_{max} and the error bound '*E'*, calculating *'t'* is straightforward.

Let $t_{diff} = p_{max} - p_{min}$. In the beginning, a node may be $(t_{diff}(1 \pm d))$ time-units out of sync with another node. After t time units, the maximum time units a node may be out of sync with another node is $(t_{diff}(1 \pm d)) \pm (2td)$. Bounding this by E, where $0 < E < 1$, we get $|(t_{diff}(1 \pm d)) \pm (2td)| < E$. Solving to get an upper bound, we get $t < (E-(t_{diff}(1+ d)))/2d$. This gives the maximum value for t after which the nodes become asynchronous and need to be re-synchronized.

2.3.3.3 Time stamping sensor readings

Time stamping helps in useful data analysis. Sensors do not have any notion of clock time, which is available on bigger computational devices – like the day and time information available on our mobile phones or PCs. However, sensors do have a timer, which can be programmed to start at some instance and then repeat after a fixed interval.

In order to time stamp a sensor's readings, we start the timer on each sensor when we are synchronizing the sensors with the RESET message. The timer is set to send an event every 100 milliseconds. Subsequently, we maintain a counter (called *'count'*) that is incremented by one, after every timer interval. This counter will give us the time relative to the timer start time. A counter value of 10 would thus mean that 10 * 100 = 1000 milliseconds have elapsed since the start of the timer. Moreover, since all the sensors start their timers at the same time (when they are synchronized) a counter value of *'c'* on two or more sensors would refer to the same time instance.

This counter value is copied into the packet sent to the base station. We realize that sending the counter value wirelessly to the base station consumes bandwidth, but it is important to do so, so that we can correlate a set of readings. It also insures that the base station only records readings in increasing order of time and drops readings that appear late. Also, it helps in evaluating when readings were lost (described later) and compare readings taken at the same time instance on different sensor nodes. We do try to save bandwidth by transmitting a single counter value for all the readings in a packet instead of sending a counter value for each reading in the packet. We are able perform the bandwidth saving because of the specific manner in which we programmed our sensors. We implemented our time stamp logic as follows:

As per our setup, we maintain 10 *'consecutive'* readings in the local array on a sensor, before sending a packet with those 10 readings to the base station. As soon as a packet is transmitted, the local array is emptied out. While recording the first value into the array, the current *'count'* value is stored in a separate variable, say *'saveCount'*. While preparing a packet to be sent to the base station, *'saveCount'* is copied into a counter field in the packet. This field gives a timestamp for each value in the packet. To understand this better, let us suppose that a packet has not been sent yet. When the first reading of that packet is being stored in the sensor's local array, we would note the counter *'count'*s current value, say '*v*', into *'saveCount'*. If the timer interval is 100 milliseconds and if '*v*' is 100, it means that the first reading of the packet was recorded 100 * 100 = 10 seconds since the start of the sensor mote's timer. Since a packet contains contiguous readings, the next reading will automatically be 101 * 100 = 10.1 seconds after the start and so on.

2.3.3.4 Data collection schemes

For our application of camera calibration, we required a simple data collection approach that would ensure the least amount of data loss in transmissions to the base station. We thus evaluated two different mechanisms for collecting data from the sensors. They are:

- *Autosend* mode: in this mode, a sensor would send a packet to the base station as soon as it had 10 readings in its local array. Ideally, a sensor would transmit a packet for the temperature readings and a packet for the luminance readings after every one second (100 milliseconds/reading * 10 readings/packet = 1 second/packet).

- *Query* mode: In this scheme, a sensor still continuously senses the environment, but it does not send data as soon as it has the fixed number of readings in its local array. Instead, the base station queries each sensor mote in a round robin fashion. When a mote receives a query message from the base station, it checks to see if a packet with 10 readings can be sent. If yes, it is sent immediately. Otherwise, it simply sets a local flag indicating that the base station query is pending. When 10 readings have been collected, if the base station query is pending, a packet with the readings is sent. There is a catch in this scheme. If a mote does not have 10 readings when the base station queried it and it sends the packet later, when the 10 readings were ready, then it may collide with another mote's transmissions. However, we try to limit such a scenario by making the base station wait for a queried node's response for a finite amount of time, before it queries the next sensor. It is hoped that a mote would have 10 readings by then.

There are reasons why we choose the above two schemes. The *'autosend'* mode is appealing because of its simplicity. Each node works as an independent unit and transmits a packet to the base station whenever a packet worth of data is ready. The base station too has no other responsibility than to collect and log data flowing towards it. However, since there is no control on any node's transmissions, there are bound to be collisions. That is the reason why we came up with a contrasting scheme – the *'query'* mode, where we can control a node's transmission to a certain extent.

2.3.3.5 Network setup evaluation approaches

There were several possible ways to set up our sensor network. However, we wanted to find the most effective sensor network setup that could be deployed to collect data for our camera calibration experiment. A network setup was considered superior to another one if it incurred fewer data losses. For this reason, we conducted several experiments, each to determine different factors and the extent to which they can lead to data losses in the network. In this section we outline our approach in estimating the data losses. We also define the granularity of the experiments we conducted.

2.3.3.5.1 Determining the amount of data lost

While comparing two or more sensor network setups, we take into account the data loss in each of them. We believe that apart from the data lost during initial startup and stabilization at each sensor node, all other data losses are attributed to collisions during transmission. A packet corrupted during transmissions is discarded at the base station.

All data losses are calculated on the PC that finally logs the data from all the sensors. A program on the PC starts tracking the losses as soon as base station sends the RESET message to synchronize and start the sensor nodes. Data loss is measured in terms of the total number of missing readings, from all the sensor nodes. Detecting losses is based on the following observation:

> If, on the PC, the previous packet from mote *'a'* had a counter value of 'x*'*, then the next packet from *'a'* should have a counter value of *'x + 10'*, since the previous packet contained 10 readings. If however, the next packet counter *'y',* is greater than *'x + 10'*, then the readings for counter values between 'y' and 'x + 10' are missing i.e., 'y – (x + 10)' readings are considered lost.

In short, the readings lost are determined by observing the counter value in a packet with respect to the previous packet's counter value from the same mote. A gap in the counter value could be due to a packet lost in transmission, readings lost on a mote itself due to factors like stabilization during initial startup. From our present understanding of the TinyOS radio and network stack, we believe that if a packet gets corrupted, then all readings are dropped. Thus, readings lost due to packet corruption through collisions will be a multiple of the number of readings in a packet (10 in our case). However, readings lost on a mote need not be a multiple of the number of readings stored in a packet.

At the end of the experiment, the data loss is calculated as a percentage of missing readings, which is the ratio of the total number of missing readings from all the motes, to the total number of (missing + correct) readings from all motes.

2.3.3.5.2 Granularity of experiments

In order to infer the best network design for an application, there are several parameters to consider and test the data losses against. Our experiments measure the data losses as a function of the following:

a. Number of sensor nodes – the number of nodes in the network is increased from 1 to 7 nodes.
b. Spatial Arrangement of the sensors. There were three mote arrangements that we tried to compare and evaluate. They were:
    - Nodes arranged in a *'straight-line'* and within 10 to 15 inches from the base station and about 3 inches from each other.
    - Motes arranged in a *'circular'* fashion around the base station with a radial distance of 10 inches between the base station and a mote. This is an arrangement where all nodes are equally close to the base station.
    - Nodes scattered in a *'random'* fashion in the laboratory – at a distance anywhere from 10 inches to 150 inches from the base station.
c. Data collection technique – autosend versus the query mode of data collection.
d. Distance from the base station – the mote distance was increased gradually up to the maximum range of transmission (about 70 feet).
e. Interference from other devices – the sensor node's transmission was tested in the presence of other devices that could possibly interfere with the sensor network transmissions.

At this stage, we have only pointed out the reasons for data losses common to the two schemes of data collection. There may be a few data loss factors specific to a particular scheme, but we will elucidate them while analyzing the results for each scheme.

We measured the data losses for varying number of motes and different spatial arrangements in conjunction with the two data collection techniques (autosend and query modes). The experiments for the last two parameters - distance from the base station and interference from other devices, were then performed only with the best data collection scheme.

2.3.3.6 Sensor network experimental results and analysis

We will first analyze the two data collection schemes individually against the three sensor node layouts we mentioned above. In each experiment, we started with one node and went up to seven nodes. The distance and interference experiments are only conducted with the best data collection scheme. It must be noted that all sensor nodes have similar battery power. All nodes were provided with fresh batteries at the start of the experiment. This is important to mention, as battery power will finally determine the strength of data transmissions and receptions.

2.3.3.6.1 Autosend Mode of Data Collection

The graph in Figure 3 shows the percentage of readings lost as a function of increasing number of motes for each of the three mote arrangements, for *'autosend'* scheme of data collection.

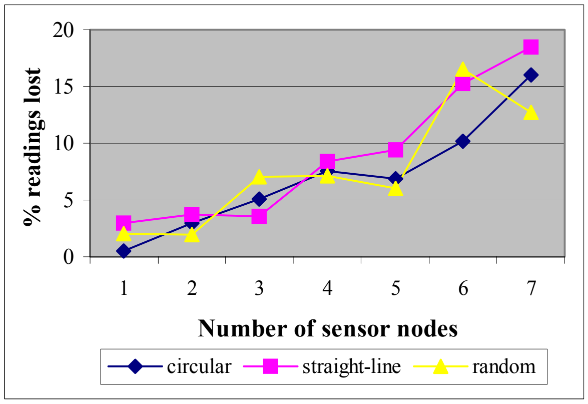

Figure 3 Percentage of readings lost in autosend mode

From Figure 3, we note that in all the sensor network layouts, in general, the percentage of readings lost increases as the number of motes increases. The increase in the percentage of readings lost is gradual in the beginning but spikes drastically as we go on increasing the number of motes. This is because, the number of transmissions increases linearly with the number of motes. When the number of transmissions goes up, so does the probability of a collision and therefore the percentage of readings lost. In some cases, we notice that the percentage of readings lost decreases by up to 4% as we increase the number of motes. We think that this could be due to the random nature of the causes behind lost readings – collisions may occur in one run of the experiment and not occur in an identical run later. Wireless telephones and other devices in the laboratory that use the same radio frequency of 916 MHz could also be the reason for this randomness. We noticed that when a phone was in use, almost no packet was successfully transmitted or received by the base station, irrespective of the number of sensor nodes in the network.

As far as the mote layout is concerned, the circular arrangement seems to give minimum data losses in almost all cases. The random arrangement gives low data losses in some cases, but in other cases gives the highest losses.

One might wonder why the data loss in autosend mode is not 100% since nodes are synchronized in the beginning and will send a packet with 10 readings simultaneously, every second. This could be credited to the CSMA [12], [13], a medium access control (MAC) protocol, which is implemented in the lower levels of the TinyOS and is responsible for transmissions. It could also be due to the loss of sync between the sensor nodes (described earlier).

2.3.3.6.2 Query Mode of Data Collection

The *'query'* mode was described in Section 2.3.3.4. We have tried various modifications of the described *'query'* mode. One of them was to allow a node to send only when the base station queries it and simply drop the query if there aren't 10 readings on the node. This would reduce the collisions that result when a node sends data out of turn. We tested this method. Since we did not get promising results from this modification, we have not included the exact results of those experiments here. It will suffice to mention that the scheme did not guarantee lower readings losses because it needs perfect synchronization between a node and the base station. If a base station queries a node when the node does not have enough readings, the node will not be able send its data until it is queried again. There is yet another option whereby we send as many readings as are available on a mote when it is queried. This too has its flaw that there will be wastage of network bandwidth and a large number of transmissions will have packets less than half full. After trying these variations of *'query'* mode, we decided to stick to the one described earlier. The base station waited for about 300 milliseconds after sending a query and before querying the next node.

Figure 4. depicts the data losses in 'query' mode. We see the same trend in data loss as we saw in the autosend mode. The losses increase as the number of nodes in the network increase. However, a closer look at the numbers show that the percentage of loss is much greater in the query mode. In the query mode, apart from readings lost due to collisions, there is another latent reason that could be causing the large number of reading losses.

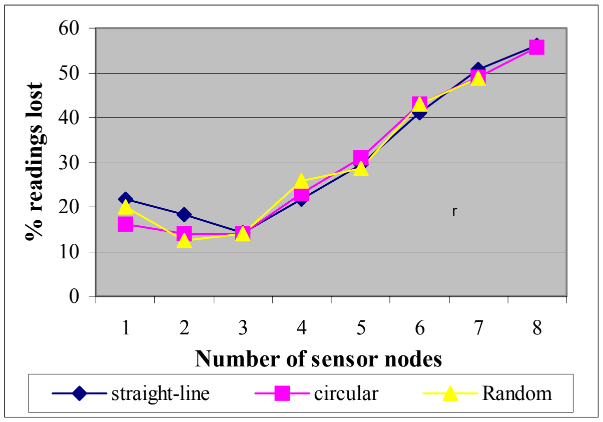

Figure 4 Percentage of readings lost in query mode

Imagine a situation in the *'query'* mode where a node has a packet with the required number of readings ready, but has not being queried by the base station. Since we have limited the amount of data stored in the local array on a node to 10 readings only, in the next timer interval, even though there is a reading to be stored, it is dropped. Readings are dropped until this mote is queried and the local array is set to blank again. The more the number of motes, the longer a particular mote will have to wait before it is queried, and thus more and more readings will be dropped. This would be a problem in any query mode operating in a continuous sensing application, unless we remove the limitation on the amount of data stored on a mote.

Another problem occurs when the base station queries a node when it has just started filling its local array. This node will not have 10 readings in the 300 milliseconds for which the base station waits, before it queries another mote. In this case, this node will send the packet later, when is has 10 readings, increasing a possibility of collision with another mote's transmission.

In query mode, there does not seem to be a clear winner among the three different mote arrangements. All the arrangements give similar results for the *'query'* scheme. It appears that the distance from the base station or the layouts do not affect the performance of the 'query' scheme as much as other reasons like wait-time before a mote is queried and collisions due to out-of-turn transmissions.

2.3.3.6.3 Autosend Vs. Query Mode

To get a clearer picture of the performance difference between *'query'* and *'autosend'* schemes, we have combined the graphs in Figure 3 and Figure 4 into one, in Figure 5. In Figure 5, all the three curves with high percentages of readings losses belong to the query mode. The percentage of readings lost in query mode is at least 10% higher than the readings lost in autosend mode. As the number of nodes increases, the losses in query mode shoot to about 50% while those in autosend stay around 15%. Thus, it is obvious that in terms of the number of readings lost, the autosend mode is far superior to the query mode.

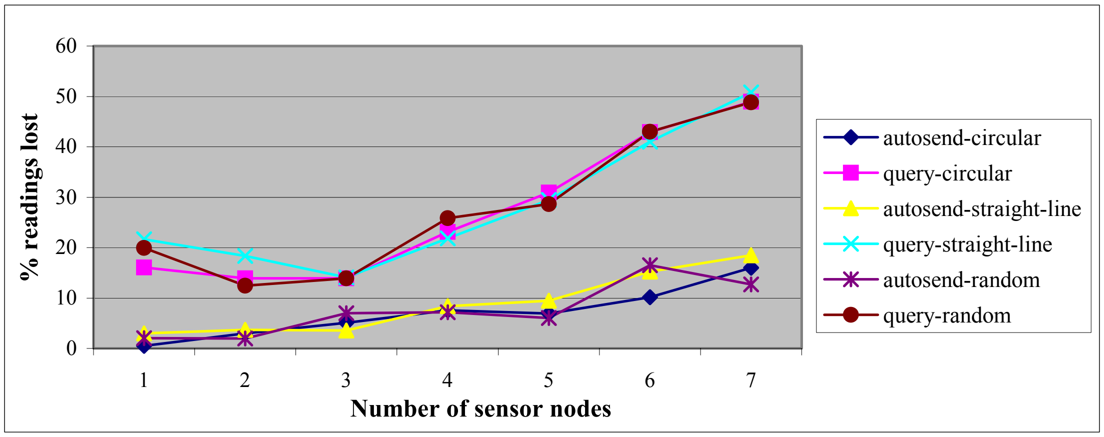

Figure 5 Percentage of readings lost in autosend and query modes

Intuitively, the query mode allows the base station to control data transmissions. The number of readings lost due to collisions should therefore be less in query mode as compared to autosend mode. However, it seems that the *'query'* mode introduces other causes for readings to get lost.

Readings are mostly lost in *'autosend'* mode due to collisions or initial startup stabilization. However, in the *'query'* scheme a large number of readings seem to be getting lost either due to collisions and lack of synchronization between the base station and the sensor nodes. These *'query'* mode problems have been discussed in detail earlier.

Another problem specific to the 'query' mode is if a node dies out in between. Since nodes are queried in a round robin fashion, the transmission slot of about 300 milliseconds for the dead node goes wasted. Not only that, at this time other motes may start loosing readings too. Such a failure has no affect on the *'autosend'* mode, but adversely affects the *'query'* mode.

The experimental results in this section showed that autosend mode is better than query mode. Hence, for the next two sets of experiments we used the autosend mode for data collection.

2.3.3.6.4 Distance from base station

We now measure the data loss as a function of increasing distance from the base station until we reach the maximum range of transmission. From our experiments, we found that the maximum transmission range was around 70 feet. In order to eliminate losses due to collisions with other sensor nodes' transmissions, in this experiment, we used only one sensor that operates in the autosend mode.

Distance from base station in Feet	Percentage of readings lost
1	2.0696617
5	2.0696617
10	3.5482258
20	1.0494742
40	4.9975259
50	3.0484757
55	5.0474762
60	1.9330504
65	1.1046817
70	100

Table 1 Percentage of readings lost as a function of increasing distances

Table 1 shows the extent to which data loss is affected by distance. We observe that in general, data losses increase with increasing distances from the base station. At 70 feet, there was 100% data loss. Large amount of material in radio-transmission literature also supports the fact the signal power decreases (losses increase) when distance between transmitter and receiver increases. Literature also suggests that many other problems, like those of *'path-loss'* and *'multi-path fading'*, start featuring with increasing distances.
There could also be random collisions with transmissions from other wireless devices (covered in next section). Due to the inter-play of multiple complex factors influencing losses, we are unable to pinpoint an exact cause for the occasional fluctuations in data losses.

This set of experiments has given us an idea of the data loss we can expect if the sensor node is far from the base station. If an application demands high data reliability and delivery, and still wants to cover a good area through its sensor network, it should consider a multi-hop network where sensors form a chain of *'adequately'* closely placed nodes, passing data to the next node which is closer to the base station, until it finally reaches the base station.

2.3.3.6.5 Interference from other devices

The Mica motes use a radio frequency of 916 MHz for wireless transmissions. In this experiment we tried to determine the extent of interference by the following devices that use radio frequencies for their operations and are commonly found in an indoor environment:

1. Telephones and wireless video transmitters: Many powerful cordless telephones and wireless video transmitters use frequencies in the 900 MHz range. We tested the performance of our sensor network in the presence of such a telephone (EnGenius SN920) and a video transmitter (accompanying CCTV-900 receivers). The results were disheartening because our sensor network had almost 100% data loss the moment these devices were switched on. These losses could not be reduced even when we tried to increase the distance between the transmitting devices (telephone and camera) and the sensors, or decrease the distance between the base station and our sensors.

2. Wireless LAN (802.11b): 802.11b wireless LANs are commonly deployed in offices. These networks use a frequency of 2.4 GHz and do not interfere with our sensor network.

3. Wireless audio transmitters: we operated our network in the presence of audio-technica's ATW-3110D audio transmitters that use frequencies between 655-680 MHz for transmitting sound information to their corresponding receivers. They did not interference with our network.

4. Other independently existing sensor motes: it is possible that there is another sensor network close to ours, which could interfere with our sensors' transmissions. We simulated such an independent network by placing some sensors in the laboratory, which would broadcast meaningless data after every 150 milliseconds. The purpose of this setup was to determine how, similarly powered devices using the same 916 MHz frequency but working independently (asynchronously), could hinder our network performance. Figure 6 shows how the data losses in our network increased as the number of sensor motes in the 'other' independent sensor network increased.

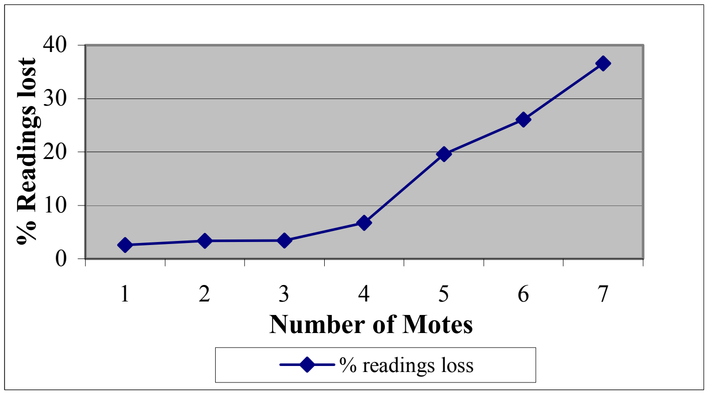

Figure 6 Percentage of readings lost as a function of number of motes in 'other' network

The graph clearly shows that the data loss is quite significant. The 'other' network is totally out-of-sync with our network and since its nodes are transmitting at a much faster rate than our node in autosend mode, maybe our node's transmissions are suffering more collisions and lesser useful data is being received at the base station.

The experiments in this category have shown that MEMS sensors' transmissions are susceptible to interference from other devices that use the same frequency range for transmissions. We have realized the Mica motes are relatively low powered as compared to bigger devices like wireless cameras and telephones. Thus, in the presence of more powerful devices, a sensor network is prone to total failure. In the presence of similarly powered devices, operating asynchronously with our network, the loss increases linearly as the number of nodes in the 'other' network goes up.

2.3.3.7 Temperature sensor inaccuracy

We noticed that even though the ambience temperature was around 21 ºC, the sensor's raw readings, when converted to engineering units (ºC) using the formula in the sensor data sheet, gave a value around 31 ºC. Moreover, even when placed under similar conditions, all sensors gave slightly different readings, about ±1 ºC around an average of 31 ºC. Correspondences with the manufacturers lead us to the understanding that the formula in the data sheet is an approximation and that each sensor should be individually calibrated. Using a thermometer of accuracy ±1 ºC, we believe that the sensor readings are offset by about 10 ºC from the actual reading. Moreover, for our experiments we did not vary the scene temperature. Thus, ideally, a sensor should give the same reading over the entire course of the experiment. On the contrary, the readings from the same sensor showed a standard deviation of about ±0.1 ºC around the mean temperature reading. These fluctuations could be due to heating of the electrical components around the thermistor. To get a reliable reading, we average the values collected from a particular sensor to get the temperature of the point where the sensor was placed.

3. Experiments for thermal IR camera calibration

We will now describe the experimental setup of the sensors and the camera in order to acquire data for the camera calibration process. The results of the experiment as well as their analysis are presented thereafter.

3.1 Thermal IR Camera

We are using the Omega thermal IR camera from Indigo Systems Corporation. The camera is linked to the computer via a 'Hauppauge WinTV' interface that allows capturing of digital images. The camera outputs gray-scaled images. The more the temperature of an object in the camera's scene, the brighter the object appears in the image. The opposite holds for colder bodies. Like the sensors, there were many issues regarding the data collection from the camera.

3.2 Experimental setup

We conducted our experiments in the indoor setting of our laboratory. Our experimental setup consisted of 8 sensors, each with a unique identification number, arranged in a grid format. The thermal camera was placed vertically above the sensor grid. The camera settings were changed in the way it is described in the previous section. We had a base station connected to a PC that collected readings from the sensors and saved them to a file on disk. Learning from our experiments discussed so far, we decided to collect data from the sensor motes using the *'autosend'* scheme. The camera periodically captured the grid covered by the 8 sensors and sent the image to the PC to be stored to disk for analysis later. The setup is illustrated in Figure 7. Examples of thermal IR image and visible spectrum image of the setup are shown in Figure 8 and Figure 9.

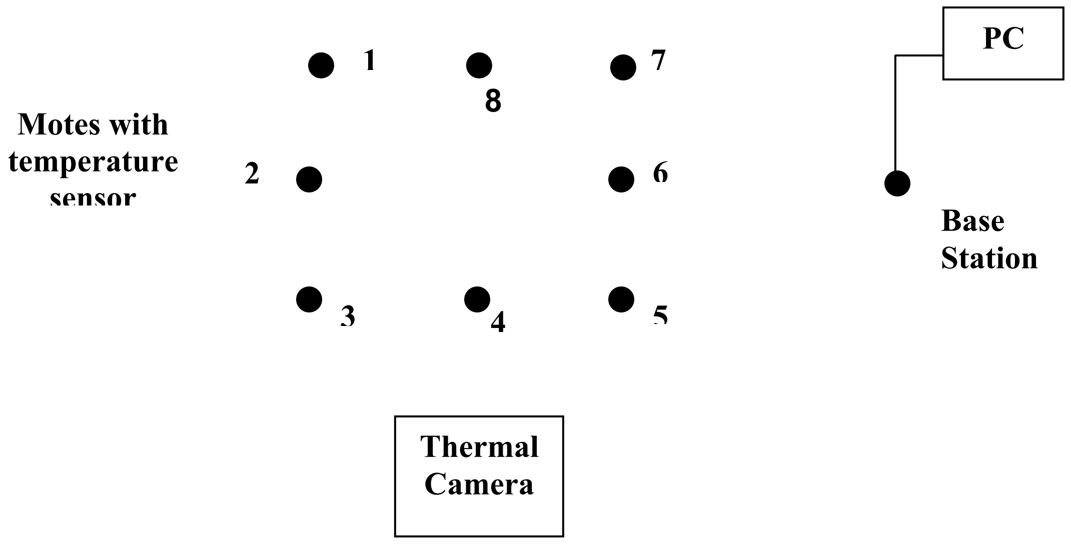

Figure 7. Experimental Setup

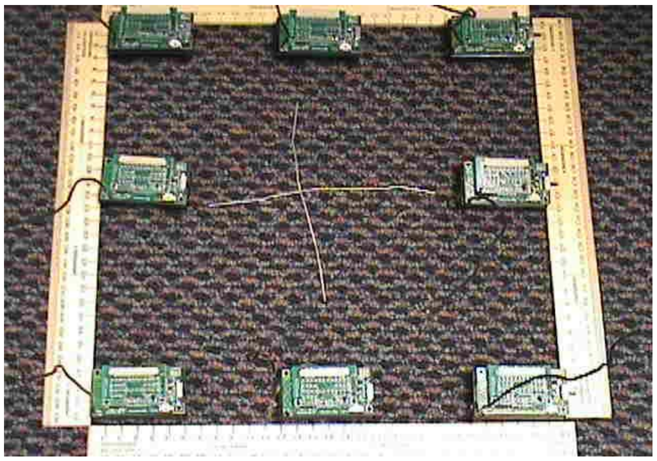

Figure 8 Experimental setup viewed through a visible camera. The tiny black spot in the middle (slightly towards the right) of a sensor mote is the thermistor, whose thermal IR pixel value we need to determine.

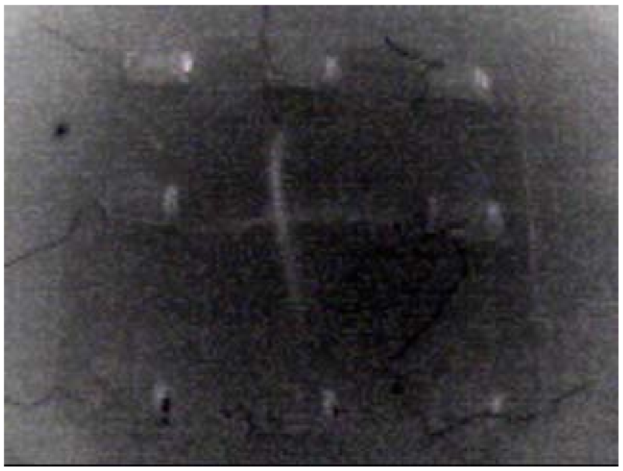

Figure 9 Temporally averaged thermal IR image

A thermometer with an accuracy of ± 1°C was also used to verify temperatures during the experiment. For the calibration experiment we kept the laboratory temperature constant at 21 °C (measured by the thermometer). The temperature close to the thermistor on each sensor was also recorded using the thermometer. The thermometer indicated the same temperature of 21 °C for all the sensors.

3.3 Preliminary calibration results

Once we have collected the data from the MEMS sensors and the camera, we can analyze them together using I2K software tools [14]. For increased reliability, we averaged multiple thermal IR images and MEMS sensor readings. These averaged images/readings are then used for calibration.

Figure 10 shows the outcome of the thermal IR camera calibration using the hot-cold pseudo-color visualization scheme. The blue color depicts the coolest regions (pixel value close to 15 corresponds to about 17ºC) while red the hottest (pixel values around 180 corresponds to about 26ºC). This image complies with our intuition. The red area is a metallic switch that is hotter than any other object in the thermal camera's field of view. With our calibration mechanism, we can assign temperature values to all the different colored regions in the image in Figure 10.

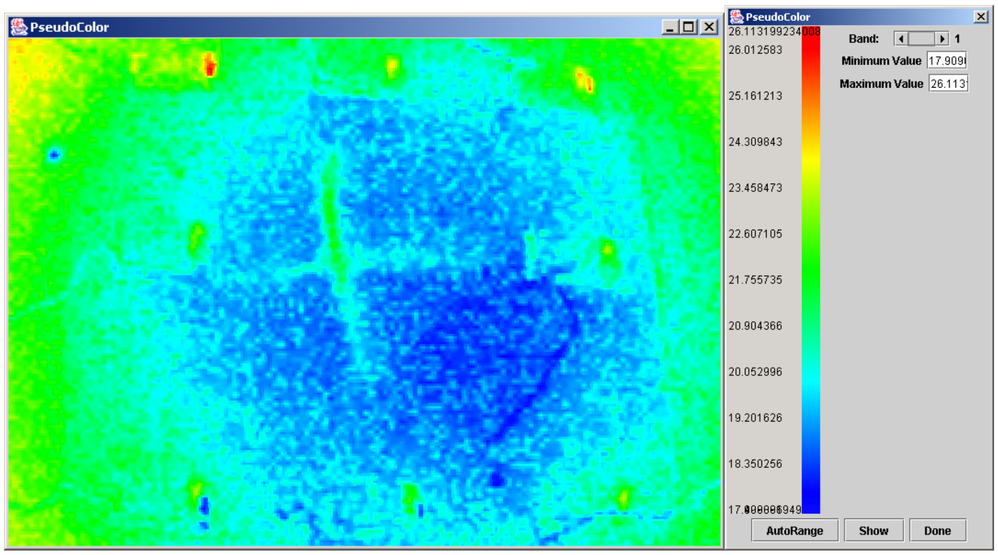

Figure 10 Pseudo-colored thermal IR image

6. Conclusion

In this report we have documented system design issues for applications that would be using wireless "smart" MEMS sensor networks. Our focus was on the spectral camera calibration application. We started by enlisting the issues related to MEMS sensors only and then highlighted some issues when the MEMS sensors and a spectral camera are integrated into a system. Finally, we explained the experimental setup and showed some preliminary calibration results for a thermal IR camera.

References

[1] http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICA.pdf
[2] Official Crossbow Inc. website. http://www.xbow.com
[3]http://www.xbow.com/Support/Support_pdf_files/MTS-MDA_Series_User_Manual_RevB.pdf
[4] TinyOS official website: http://webs.cs.berkeley.edu/tos
[5]MICA: The Commercialization of Microsensor Motes http://www.sensorsmag.com/articles/0402/40/
[6] Hanbiao Wang, Deborah Estrin and Lewis Girod: "Preprocessing in a Tiered Sensor Network for Habitat Monitoring.
[7] Alan Mainwaing, Joseph Polastre, Robert Szewczyk, David Culler and John Anderson: "Wireless Sensor Networks for Habitat Monitoring".
[8] Raghavendra C.S and Suresh Singh: "PAMAS – Power Aware Multi-Access protocol with Signalling for Ad Hoc Networks"
[9] Wei Ye, John Heidemann and Deborah Estrin: "An Enery-Efficient MAC protocol for Wireless Sensor Networks"

[10] Jeremy Elson and Deborah Estrin. "Time Synchronization for Wireless Sensor Networks". In 2001 International Parallel and Distributed Processing Symposium (IPDPS), 2001.
[11] J. Elson, Lewis Girod and D. Estrin. "Fine-Grained Network Time Synchronization using Reference Broadcasts". Submitted for review, February 2002.
[12] L. Kleinrock and R. Tobagi. Packet switching in radio channels, part 1: Carrier sense multiple-access models and their throughput-delay characteristics.
[13] S.S. Lam. " A carrier sense multiple access protocol for local networks." In *Computer Networks, volume 4,* pages 21 – 32, 1980.
[14] Bajcsy P., "Image To Knowledge", documentation at web site:
http://alg.ncsa.uiuc.edu/tools/docs/i2k/manual/index.html

<u>Appendix A</u>

Troubleshooting some common problems:

- ***Programming the motes***
  In order to program a mote with an application, follow the steps given below:

  1. On the computer configured with TOS tools, compile the TinyOS application code that you want to program your mote with. If you have successfully installed TinyOS then if are in the directory where the code resides, typing "*make mica*" compiles the code.
  2. Place the mote board (or the mote and sensor stack) into the bay on the programming board. In order to programme a board, one must supply a 3-volt supply to the connector on the programming board or power the node directly. The red LED on the programming board will be on when power is supplied.
  3. Plug the 32-pin connector on the programming board into the parallel port on the computer that has the TinyOS installed, using a standard DB32 parallel port cable.
  4. Type: *"make mica install*". This should install the code on the mote. If you want to designate an identification number (ID) of the mote, type the ID number after install. For example, typing "*make mica install.2*" will assign an ID of 2 to the mote.
  5. On the computer configured with TOS tools, compile the TinyOS application code that you want to program your mote with. If you have successfully installed TinyOS, then if you are in the directory where the code resides, typing "*make mica*" compiles the code.
  6. Place the mote board (or the mote and sensor stack) into the bay on the programming board. In order to programme a board, one must supply a 3-volt supply to the connector on the programming board or power the node directly. The red LED on the programming board will be on when power is supplied.
  7. Plug the 32-pin connector on the programming board into the parallel port on the computer that has the TinyOS installed, using a standard DB32 parallel port cable.

8. Type: *"make mica install*". This should install the code on the mote. If you want to designate an identification number (ID) of the mote, type the ID number after install. For example, typing "*make mica install.2*" will assign an ID of 2 to the mote.

- ***Generating documents for debugging***
  A useful feature in TinyOS is that it allows you to generate documentation on the fly, during code compilation. The document thus generated gives a pictorial representation of how the different components used by the code are wired to each other. This can be very helpful during debugging as it allows the programmer to ensure that the components are linked as intended by the programmer. One can generate the document for a code by typing: "*make docs mica*" for compiling the code instead of typing "*make mica*".

- ***Motes programmed without errors, but not giving the desired output?***
  Before you do anything else, make sure you know your sensors and Mica board well (http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/micasbl.pdf) and have checked some of the files that are used during installation of your program on the motes. One such file is the *Makerules* file in the tinyos-1.x/apps directory. By default, when we say *'make mica install'* while programming our motes, Makerules file assumes that the mica board is of type '*micasb*'. However, as we found out later, after a few weeks of futile effort at programming our motes to give the right output, our motes were of type '*basicsb*'. To get the right output, we then changed the following Makerules file's lines:

```
#Sensor Board Defaults
ifeq ($(SENSORBOARD),)
    ifeq ($(PLATFORM),mica)
                #    SENSORBOARD = micasb  …comment this out
            SENSORBOARD = basicsb  …add this line
    endif
```

  If you are sure that the problem is not in your code, and that all the components are wired well in your code's configuration file, then you should check the other ancillary tinyos files to ensure that the right setting are being used. Directory contents of '*C:\tinyos-1.x\tos\sensorboards'* give the detailed pin layout of different Mica boards and comparing them with your own board will help to determine your board.

- ***Sometimes one can't obtain data from the ADC component on the motes.***
  ADC component can handle only one request at a time. So, if you call ADC.getData() successively, before the first call has been fulfilled(through the signaling of the ADC.dataReady() event), the second call will fail.