An Intelligent Approach for Deploying Applications in WSN's

Dimitrios Georgoulas

Doctor of Philosophy

ASTON UNIVERSITY
February 2009

ASTON UNIVERSITY

An Intelligent Approach for Deploying Applications in WSN's

Dimitrios Georgoulas

Doctor of Philosophy, 2008

Thesis Summary

Wireless sensor networks have been identified as one of the key technologies for the 21st century. They consist of tiny devices with limited processing and power capabilities, called motes that can be deployed in large numbers over a physical environment providing the end user with a number of useful sensing capabilities. Even though, they are flexible and easy to deploy, there are a number of considerations when it comes to their fault tolerance, conserving energy and re-programmability that need to be addressed before we draw any substantial conclusions about the effectiveness of this technology.

In order to overcome their limitations, we propose a middleware solution. The proposed scheme is composed based on two main methods. The first method involves the creation of a flexible communication protocol based on technologies such as Mobile Code/Agents and Linda-like tuple spaces. In this way, every node of the wireless sensor network will produce and process data based on what is the best for it but also for the group that it belongs too.

The second method incorporates the above protocol in a middleware that will aim to bridge the gap between the application layer and low level constructs such as the physical layer of the wireless sensor network. A fault tolerant platform for deploying and monitoring applications in real time offers a number of possibilities for the end user giving him in parallel the freedom to experiment with various parameters, in an effort the deployed applications to run in an energy efficient manner inside the network.

The proposed scheme is evaluated through a number of trials aiming to test its merits under real time conditions and to identify its effectiveness against other similar approaches. Finally, parameters which determine the characteristics of the proposed scheme are also examined.

Keywords: Wireless Sensor Networks, Middleware, Intelligence, Mobile Code, Tuple Spaces

To my parents, Konstantinos and Aggeliki, t

my friend, Liann

Table of Figures

List of Tables

Acknowledgements

I wish to thank Professor. Keith B. for inspiring me to carry out this research under his supervision. His great support, suggestions and feedback were the most basic ingredients for the completion of my research. I also wish to thank Dr. David H. for giving very useful feedback on various aspects of my research topic.

At this point I would like to thank the Engineering and Physical Sciences Research Council (EPSRC) for granting me a research studentship, which covered the tuition fees of this course. I am also very grateful to the senior administrator, Miss Helen Yard and to the Research Student Coordinator Ms Sharen Lloyd for both being extremely helpful and for dealing with all the administrative issues that arose within this time period.

Last, but not least, I would like to thank all my friends and colleagues who made my 3-year stay as pleasant as possible and especially Dr. X. Peng, Dr. S. Fowler, Dr. Kostis Koumpis, Dr. Alexandros Papanikolaou, Qasim Iqbal, Steve Green, Thom Hill, Ritesh Taank, Richard Haywood, Satty Mukherjee, Guchun Zhang, JooBee Yeow, Annita Lawlor, Alicia Morris, James Gormley and Eve Tsetsou.

Publications Related to the Thesis

[1] Dimitrios Georgoulas and Keith Blow. Making Motes Intelligent: An Agent-Based Approach to Wireless Sensor Networks. *International Journal on Communications*, pp 515-522, 2006, WSEAS.

[2] Dimitrios Georgoulas and Keith Blow. In-Motes: An Intelligent Agent Based Middleware for Wireless Sensor Networks. In SEPADS, *in proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, WSEAS, pp225-231, Madrid, Spain, February 2006.

[3] Dimitrios Georgoulas and Keith Blow. In-Motes Bins: A Real Time Application for Environmental Monitoring in Wireless Sensor Networks. In IEEE, *in proceedings of the 9th IEEE/IFIP International Conference on Mobile and Wireless Communications Networks*, pp21-26, Cork, Ireland, September 2007.

[4] Dimitrios Georgoulas and Keith Blow, Intelligent Mobile Agent Middleware for Wireless Sensor Networks: A Real Time Application Case Study. In AICT 2008, *in proceedings of 4th Advanced International Conference on Telecommunications: Programmable networks, active networks and mobile agents, protocol & standards*, Athens, Greece June 2008.

Awards Related to the Thesis

[1] The *Best Student Paper Award* of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems, SEPADS 06, was given to Dimitrios Georgoulas for his paper: Making Motes Intelligent: An Agent-Based Approach to Wireless Sensor Networks.

[2] The *Best Student Project of the Year 06 Award* of Computing Awards for Excellence was given to Dimitrios Georgoulas for his research: In-Motes: An Intelligent Agent Based Middleware for Wireless Sensor Networks/In-Motes Bins Application

[3] The *Best Student Project of the Year 07 Award* of Computing Awards for Excellence was given to Dimitrios Georgoulas for his research: In-Motes Reloaded middleware /In-Motes EYE Application

CHAPTER 1

Introduction

1.1 Motivation

Wireless sensor networks are one of the key promising technologies for the 21st century (Alkyildiz et.al 2002). They consist of tiny devices, with limited computation abilities, that can co-operate their actions over a wireless link and provide a number of sensing modules. The power of this technology lies in the fact that those devices, motes, can be deployed in a large scale over an environment without the constant attention of an end user. Usage scenarios for these devices range from geophysical monitoring to real time tracking, to military applications and in the near future they may even incorporated to our everyday lives with health monitoring applications leading currently the way.

The most straightforward application that someone can think of applying this technology is in the area of environmental monitoring. With a low cost distributed network and a few hundreds of sensors, an area even in the hardest physical location could be monitored providing to the end user useful information about the progress of a certain harvest. "*...Imagine smart farmlands where literally every...vine plant will have its own sensor...making sure that it gets exactly the right nutrients, exactly the right watering. Imagine the impact it could have on difficult areas of the world for agricultural purposes.*" Intel Chief Technology Officer Pat Gelsinger said when he was asked about the future plans of their project, wireless vineyard, Figure 1-1 (Intel Magazine 2006).

Figure 1-1: Part of the vineyard with sensors placed by Intel researchers across the monitoring area www.intel.com/research/vert_agri.htm

Currently, wireless sensor networks are beginning to be deployed at an accelerated rate with new promising projects, both academic and industrial, appearing in a monthly base. Thus, it is not unreasonable to expect that in 10-15 years the world will be covered with wireless sensor networks with access to them via mediums such as the Internet. This new technology is exciting with unlimited potential for numerous application areas as we mentioned above. In order for the wireless sensor dream to become a reality, there are a number of limitations and hurdles that need to be addressed and resolved.

Most of the sensor devices such as the Berkeley mica2 motes, Figure 1-2, are battery supplied, instantly raising the issue of energy conservation in a wireless sensor network. The life expectancy of the network is tightly bound with the life expectancy of its nodes which in terms is regulated by the energy levels that these batteries are providing. Thus, a mechanism that will regulate the activity pattern of each node inside the wireless sensor network is important, as is equally important the generation of a scheme that will try to conserve as much energy as possible.

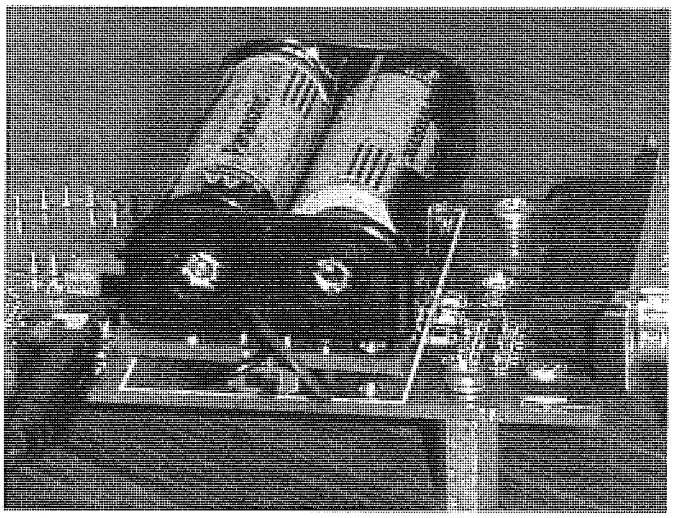


Figure 1-2: A mica2 mote attached on a MIB510 programming board, two AA batteries are its only power source once it's been deployed

One of the main advantages of the wireless sensor networks, as we already mentioned, is the easiness in deployment of a vast number of wireless tiny sensor nodes. Without though the proper mechanisms for monitoring and controlling a network of such a scale, problems such as package loss and buffer overflows can easily lead to catastrophic scenarios where the whole network will stall its operation. A reliable communication protocol and a strong hierarchy are a necessity, in order that complicated interconnections between the wireless nodes and the unreliable radio do not lead to network failures and also the wireless sensor network applications should have a clear scope inside their activity range.

Last, but not least, the tiny sensor devices usually are pre-programmed with a specific application prior to any deployment. For example, mica2 sensors need to be attached to an interface programming board such as the MIB510, and then connected to the user's terminal in order for a certain module or an application to be uploaded.

Thus, in most cases post development re-programming is not feasible, as it would imply that a programmer must access the environment, collect all the sensor nodes, re-program them and re-deploy them while in parallel making sure that such interference has not altered any of the experimental parameters of the environment. A mechanism that would provide the user with the freedom, to alter individual parameters for each sensor such as bandwidth and wake up calls and even re-install part of the application remotely incase of an unexpected error, is of great importance.

1.2 Aims of the thesis

1. To investigate how we can increase the reliability, fault tolerance and flexibility of a wireless sensor network by focusing mainly on the application layer of these networks.
   (a) Investigate the wireless sensor network theory and identify key limitations of this technology
   (b) Identify architectures and approaches in wireless sensor networks by categorizing them in terms of functionality and organization
   (c) Focus on the application layer and investigate the middleware approach as a viable solution for overcoming the limitations that the wireless sensor networks inhabit

2. To develop a method that will be able to moderate effectively the interconnections between the wireless nodes and that will provide recovery mechanisms incase of a failure.
   (a) Devise a dynamic communication protocol with the means of:
   I. Mobile code technology and some axioms of agent technology

II. Linda like tuple spaces

III. Behavioral rules based on a model of bacterial strains

3. To develop an energy efficient method that will be able to provide remote mechanisms for controlling and monitoring the wireless sensor network during the lifetime of an application.

   For the above methods, it is also required:

   (a) To develop a prototype framework for deploying applications in a wireless sensor network

   (b) Incorporate the communication protocol in the system (s)

   (c) To develop a number of prototype applications for experiments and testing in order that the end system can be evaluated.

   (d) To develop a comparison model for evaluating our approach against a number of other approaches that exist in the same category.

More analytically, the first aim of the thesis requires a critical investigation in the wireless sensor network literature in order to identify key methodologies and theories behind this technology. What also needs to be determined is the main organization and functionality of those networks. The organization will provide us with a number of controlling and monitoring methods that will enable us to understand better the nature and applicability of the network. Regarding the functionality review, this will reveal us a number of approaches and architectures aiming to provide attributes such as fault detection and managing power resources at an application level.

As far as the second aim is concerned, it should be determined whether certain interconnections between the wireless nodes can affect the network performance. Should this prove to hold, a communication protocol should be developed incorporating all those parameters and aiming to normalize the performance by providing fault detection and recovery mechanisms.

The last aim deals with the creation of a powerful framework that will try to address and resolve problems in the wireless sensor networks such as the ones mentioned above. It is quite obvious that it should provide an energy efficient way for deploying applications in a wireless sensor network and a flexible mechanism for collecting readings from the nodes. Also, it should be able to incorporate in its infrastructure the above communication protocol and be able to react in unexpected scenarios in an efficient manner. For evaluating its applicability and correct functionality a number of trials should be generated identifying possible development and application errors. A comparison model for evaluating our method against others in the same domain is necessary in order to identify the strengths and the weaknesses of our approach. Ideally, real time trials in a dynamic environment should be carried out through complicated applications, helping us understand better both the nature of the deployed network as well as the effectiveness of our proposed method.

1.3 Research methodology

1.3.1 Literature review

- Wireless sensor networks

We investigate the wireless sensor network theory and we categorize those networks in terms of functionality and organization. A number of system architectures such as MANNA (Ruiz et.al 2003), WinMS (Lee et.al 2006) and MOTE-VIEW (Touron 2005) are critically reviewed. Finally, we focus on the middleware role in the wireless sensor networks by identifying how it can establish a framework for bridging the gap between applications and low levels constructs such as the physical layer of the sensor nodes. A number of design parameters and considerations are elaborated together with the requirements of a flexible wireless sensor network, in an effort to identify how this approach can resolve the limitations that those networks inhabit. Last, but not least, we present a classification and analysis of current middleware approaches since they are the foundations of our proposed approach.

- The key methodologies of our approach

Rather than re-inventing the wheel, for our proposed approach we use a number of well established architectures and technologies that we bound together under the same framework. Thus, we review the mobile code/agent technology and we define what we name as In-Motes agent inside our wireless sensor network, reflecting in parallel concepts that we incorporate in our mobile code framework for the development of the system (s). The communication methodologies for those entities are presented as well. The concept of network management based on an analogy with bacterial strains is analyzed since it has been elaborated in the communication protocol we developed. Lastly, the TinyOS operating system key terminologies are explained in order to understand better the medium between our approach and the hardware we are using.

1.3.2 Prototyping

Two prototype middlewares were developed and implemented namely the In-Motes and the In-Motes Reloaded (Georgoulas & Blow 2006), for carrying out our proposed approach using a combination of nesC and Java programming languages. Their design and coding was based on the key methodologies we mentioned above and were aiming to overcome in the most reliable way the limitations that the wireless sensor network proved to inhabit. The In-Motes Reloaded is an updated version of the In-Motes middleware and it was created based on the lessons learned after a four month trial of In-Motes in a dynamic real time environment.

1.3.3 System evaluation

A series of experiments and trials were followed, for testing and evaluating the functionalities of both middlewares. Initially, all the tests were carried out in the laboratory with simple applications and small scale wireless sensor networks. The information gathered was then cross checked with the information gathered when the middlewares were tested under real conditions in dynamic environments by deploying more complex applications in larger scale networks. For the In-Motes middleware we developed the In-Motes Bins application that was a mobile code based application for monitoring light and temperature variations in a wine store for a period of four months. In-Motes EYE was deployed from the In-Motes Reloaded middleware and it was the second complex application that was developed for monitoring the acceleration of moving objects. Graphical representations of the results were used in all the scenarios in order to visualize the experimental data and the performance of the middlewares with respect to certain parameters.

1.4 The novel aspects of the thesis

The novel aspects of this thesis are:

- A method for making wireless sensor nodes highly interactive and application specific in a given environment by incorporating (Chapter 4):
    - ✓ Mobile code technology
    - ✓ A federated architecture for the software communication
    - ✓ Linda-like tuple spaces architecture for the hardware communication
    - ✓ Behavioral rules based on bacterial strains for added hierarchy
- A method for deploying applications in the wireless sensor network in an efficient manner in terms of conserving energy, re-programmability and fault tolerance by developing (Chapters 4/6):
    - ✓ The In-Motes middleware using nesC and Java programming tools
    - ✓ The In-Motes Reloaded middleware which is an updated version of In-Motes with added flexibility and a more efficient programming engine (better resource management, less run time errors)
- A method for monitoring environmental conditions in a dynamic everyday environment by developing (Chapters 5/6):

- ✓ The In-Motes Bins application which stands as a multi-hop mobile code based In-Motes application for measuring variations in light and temperature
- ✓ The In-Motes EYE application which stands as a multi-hop mobile code based In-Motes Reloaded application for measuring variations in the acceleration of moving objects

1.5 The outline of the thesis

The outline of the thesis is organized as follows: Chapter 2 presents the fundamentals of wireless sensor network, thus providing the necessary background required for understanding the organization, functionality and limitations of those networks. The middleware solution is also investigated through a critical presentation and analysis of some of the most well established approaches. Chapter 3 presents the key methodologies that were used for the development of our proposal, thus providing the foundations of our research approach. Technologies, communication schemes and architectures such as the Mobile code/Mobile Agents and the Linda-like tuple spaces are explained. Last, but not least, we explain and reason the programming tools we used for the development of our middleware. Chapter 4 describes the design of the In-Motes middleware. All the communication and architectural concepts are presented and explained together with an analysis of the low level design of In-Motes core programming engine. In Chapter 5 the evaluation of the In-Motes middleware is taking place through a series of tests aiming to identify the merits of our approach. Preliminary laboratory tests together with the In-Motes Bins application are explained by using a series of graphs together with a list of lessons learnt for each experimental procedure. The In-Motes Reloaded middleware is presented in Chapter 6 as an updated version of In-Motes. The main modifications of the new approach are explained followed by an evaluation trial of the system through the In-Motes EYE application. Finally, the conclusions and proposals for future research are presented in Chapter 7.

CHAPTER 2

Wireless Sensor Networks

2.1 Introduction

Wireless sensor networks have been identified as one of most important technologies for the 21st century (Culler et.al 2004). As technologies advance and hardware prices drop, wireless sensor networks will find more prosperous ground to spread in areas where traditional networks are inadequate. The foundational concept which applies in a vast number of networks can be identified through the simple notion: Sensing Capabilities plus CPU Power plus Radio Transmission equals A powerful framework for deploying thousands of potential applications.

However, this notion is underlined by some complex and detailed understanding of each separate network components capabilities and limitations as well as understanding in areas of modern network management and distributed systems theory.

The primary goal of wireless sensor networks is to make useful measurements for as long as possible. To do this it is essential to minimize energy use by reducing the amount of communication between nodes without sacrificing useful data transmission. Each node is designed in an interconnected web that will grow upon the deployment in mind. Wireless sensor networks are highly dynamic and susceptible to network failures, mainly because of the physically harsh environments that they are deployed in and connectivity interruptions (Hill et.al 2000).

To make the wireless sensor network dream a reality, an architecture must be developed that will monitor and control the node communication in order to optimize and maintain the performance of the network, ensure that the network operates properly and control/instruct a set of cluster nodes without human intervention (Boulis et.al 2003).

In order to develop a system architecture with the above characteristics, we focus explicitly on the functions and the roles of wireless network management systems. Additionally, we present the middleware concept as a novel solution to the limitations that wireless sensor networks inhabit. A number of network systems are presented, critical reviewed and categorized providing the motivation and support for the design choices of our system that will be presented in the next chapter.

2.2 Network management systems organization and functionality

Around 1980s computer networks began to grow and be interconnected in a large scale. This growth produced problems in maintaining and managing those networks, thus the need of network management was realized. Today, networks are far more dynamic and interconnected than before, especially in the area of wireless sensor networks, thus a managing infrastructure is one of the most basic requirement for monitoring and controlling such networks (Cisco Systems 2000).

A network management system can be defined as a system with the ability to monitor and control a network from a central location. Ideally there are four key functional areas that this system must support (Stallings 2004):

Fault Management: This area provides the facilities that allow the discovery of any kind of faults that the managed devices of the network will produce, determining in parallel the possible causes of such errors. Thus, the fault management function should provide mechanisms for error detection, correction, log reports and diagnostics preferably without the user interference.

Configuration Management: Responsible for monitoring the entire network configuration information and also having access to all the managed devices in terms of reconfigure, operate and shut down if necessary.

Performance Management: Responsible for measuring the network performance through analysis of statistical data about the system so that it may be maintained at an acceptable level.

Security Management: This area provides all those facilities that will ensure that access to network resources can not be obtained without the proper authorization. In order to do so, it provides mechanisms for limiting the access to network resources and provides the end user with notifications of security breeches and attempts.

Those four functional areas of network management are far more challenging and vital for a network that will consist of tiny sensors which can be supplied to a specific environment running applications such as habitat monitoring, microclimate research, medical care and structural monitoring (Culler et.al 2004). For every sensor network application, the network is presented as a distributed system consisting of many autonomous nodes that cooperate and coordinate their actions based on a predefined architecture. Every node is assigned with a specific role inside the network such as data acquisition and processing. Also, nodes can be used as data aggregation and caching points in order to reduce the communication overhead (Akyildiz et.al 2002).

2.2.1 The system organization of a sensor network system

The organization of sensor network systems is based upon the approach that they will adapt in order to monitor and control the state of the wireless sensor network. There are four predominant approaches (Lee et.al 2003):

Passive Monitoring: The system role is to collect data during the lifetime of the network. The data will identify the state of the network in different time intervals without any action taking place during the data gathering. An analysis of the data will take place in later stages.

Fault Detection Monitoring: The system dedicates its resources to identifying faults and errors during the lifetime of the network. All the information is gathered and reported back to the operator whose responsibility is to correct those problems in later stages. No action is taken by the system towards the resolution of those problems in real time.

Reactive Monitoring: The system has a double role to accomplish during the lifetime of the network. Firstly, as we identified in the previous approaches, the collection of data that will provide information about the states of the network, is the main role. This time though, the system will be eligible to identify and detect any events and act upon them in real time mainly by altering the parameters of the fixed asset under its control.

Proactive Monitoring: The system collects and analyzes all the incoming data concerned with the state of the network. Then an analysis is taking place similar to the one of the reactive monitoring with the big difference that certain events, described by the collected information, are stored. The system is then able to maintain better available network performance by predicting future events based on past ones.

Wireless sensor systems can be categorized according to their architecture which can be centralized, hierarchical or distributed (Akyildiz et.al 2002). The centralized one identifies the role of the base station as the most important one in the whole architecture. The base station will collect the information from all the nodes and will monitor and control the entire network. Benefits to this architecture can be found in areas of processing power and decision making. A base station with unlimited power resources can perform complex analysis of data and process a variety of information, reducing the weight of this energy consuming task from the nodes of the network.

The distributed architecture focuses on the deployment of multiple manager stations across the network usually in a web based format. Thus, each substation can coordinate its actions and co-operate based on knowledge that it can acquire from a neighboring substations. In this approach, the communication cost is less than the centralized one and more energy efficient since all the workload will be distributed evenly across the network. However, due to the scalability and complexity of wireless sensor networks it is proven quite difficult to manage and quite expensive in terms of memory cost.

The hybrid between the centralized and distributed approach is that of the hierarchical one. In this architecture we have the existence of substations in the network but this time no communication is allowed between them. The design tends to be cluster based, with the heads of the cluster be responsible for a set of network nodes in terms of processing and transmitting information. All the cluster heads will report back to a single base station.

2.2.2 The functionality of sensor network systems

The main functionality of sensor network systems is based on the theory behind network management systems, thus is focusing on two attributes those of monitor and control. In this section, we classify some well known sensor systems in terms of the functionality they provide inside the network. Figure 2-1 is demonstrating this classification.

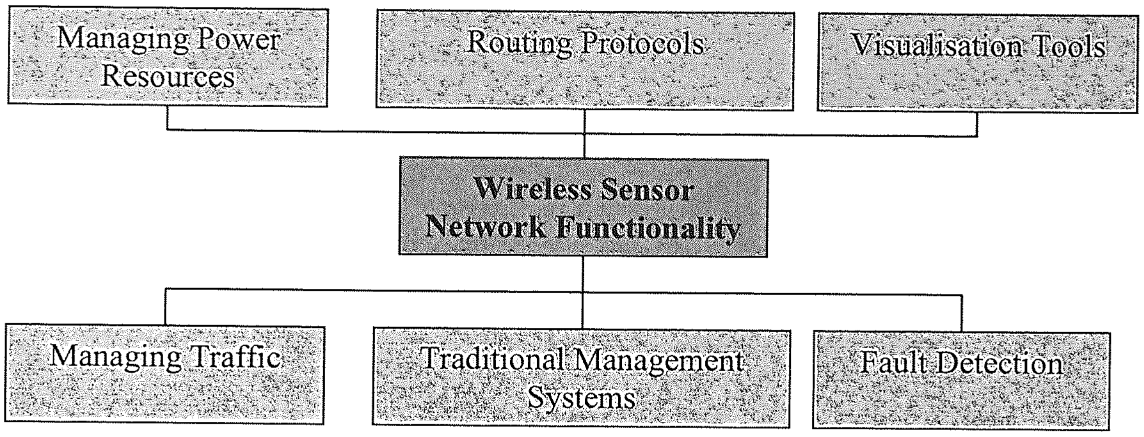


Figure 2-1: The Functionalities of Wireless Sensor Networks in different categories

Two characteristic examples of wireless sensor networks that are based on traditional management systems are those of MANNA (Ruiz et.al 2003) and BOSS (Song et.al 2005). MANNA provides a general architecture for managing a wireless sensor network by using a multidimensional plane for the functional, physical and the informational architecture of the network. The functional plane is responsible for the configuration of the application specific entities, the information plane is object oriented and specifies all the syntax and semantics that will be exchanged between the entities of the network and lastly the physical plane establishes, according to the available protocols profiles, the communication interfaces for the management entities that will be present inside the network.

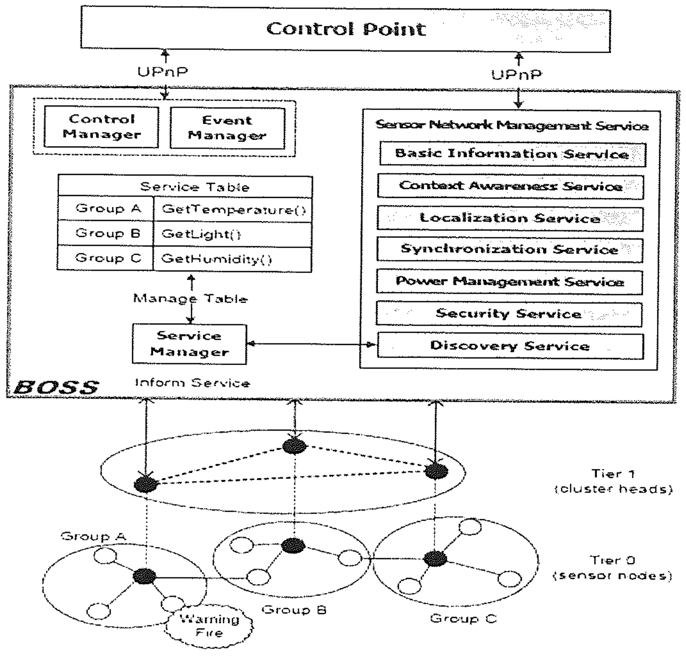


Figure 2-2: The BOSS architecture, Song et.al 2005

The BOSS architecture, Figure 2-2, is based on the traditional method of the standard service discovery protocol, UpnP. With the UpnP protocol the user does not need to self configure the network and devices in the network automatically will be discovered. However, due to computational power consumption required by the devices and memory space allocation limitations, the protocol itself is not suitable for tiny sensor devices. BOSS architecture is overcoming this problem by acting as a mediator between UpnP networks and sensor nodes. In order to do that, it combines four different components: service manager, control manager, service table and a sensor network management service, under the same framework.

Routing protocols is another alternative way of monitoring and controlling a wireless network when they get embedded in an application with examples such as LEACH (Heinzelman et.al 2000) and GAF (Xu et.al 2001).

LEACH is a routing protocol for users that want remotely to monitor an environment. The protocol is build upon two foundational assumptions. The first one acknowledges that the base station is at a fixed point and in a far distance from the network nodes and the second one assumes that all nodes in the network are homogeneous and energy constrained. In order to maximize the system lifetime and coverage, LEACH is using a set of methods such as distributed cluster formation with randomized selection of cluster heads and local processing. LEACH dynamic clustering method, splits time in fixed intervals with equal length. Also, it does not allow clusters and cluster heads to be at a fixed point inside the network. LEACH, dictates that once other sensors of the network receive a message they will join a cluster with the stronger signal cluster head.

GAF, which stands for geographic adaptive fidelity, focuses its architecture on the extension of the lifetime of the network by exploiting node redundancy. This node redundancy is achieved by switching off unnecessary sensor nodes in the network without any effect on the level of routing fidelity.

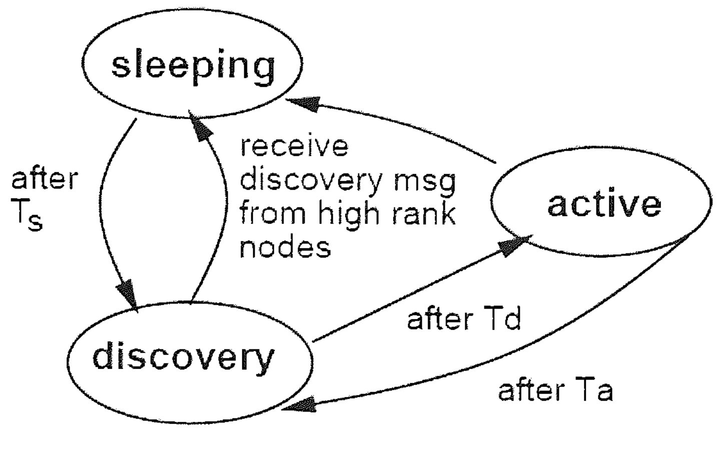


Figure 2-3: The GAF nodes state transitions, Xu et.al 2001

GAF recognizes three transition states for the nodes, Figure 2-3, active, sleeping and discovery. Initially all nodes in the network are in a discovery state. This means that all nodes will turn their radio on and exchange discovery messages in order to identify neighbor nodes in the same grid. When a node is active it will set a timeout, Ta, that will determine for how long it will stay in that state before it returns back to the discovery state. While active, the node periodically re-broadcasts its discovery message at time intervals, Td. The sleeping state is regulated by a time interval Ts which is dependent upon the application. GAF assumes that sensor nodes can identify their location in the forming virtual grid with the use of GPS cards.

Systems, such as WinMS (Lee et.al 2006) and Sympathy (Ramanathan et.al 2005) focus more on the importance of fault detection in a wireless sensor network. WinMS uses a novel management function, called systematic resource transfer, in order to provide automatic self-configuration and self-stabilization both locally and globally for the given wireless sensor network. This function allows the network, in case of a failure, to have a predetermined period of time where nodes will listen to their environment activities and self-configure. No prior knowledge of the topology of the network is necessary. WinMS, uses a TDMA-based MAC protocol, called FlexiMAC (Lee et.al 2006), in order to support resource transfer among nodes in the network. FlexiMac protocol provides synchronized communication between the nodes. Thus, it can adaptively adjust the network by providing local and central recovery mechanisms.

Sympathy is a tool for detecting and debugging failures in wireless sensor networks, but unlike WinMS it does not provide any automatic network reconfiguration incase of a failure. One of the main functionalities of the system is the collection and analysis of network information metrics such as nodes next hop and neighbors. By doing so, it is able to identify which of the nodes deliver insufficient data to the sink node or to the base station and locate the cause by reporting back to the end user. One of the major advantages of Sympathy is that it takes into account interactions upon multiple nodes however, by doing so it will require nodes to exchange neighborhood lists, something that has proven highly costly in terms of energy levels.

Another very useful functionality of wireless sensor systems is that of the visualization of the actual network. This ability of an end user to demonstrate graphical representations of the different states of the network at various time intervals can be found in systems such as the TinyDB (Maden et.al 2005) and MOTE-VIEW (Touron 2005).

TinyDB is a distributed query processor for sensor networks. It uses an SQL like interface for collecting data from nodes in the given environment and also provides aggregation, filtering and routing of the acquired results back to the end user. With the use of a declarative language for specific user queries, TinyDB proves to be flexible in two domains. Firstly, all the queries that are generated are easy to read and understand and secondly the underlying system will be responsible for the generation and the modifications of the query without the query itself to need any modifications. In the core of the system we find a metadata catalog that identifies the commands and attributes that are available for querying.

The MOTE-VIEW system is an interface system between the end user and the deployed network of wireless sensors. Through this interface the user can make alterations to node characteristics in terms of radio frequency, sampling frequency and transmission power. The system architecture is based on four layers; data access abstraction, node abstraction, conversion abstraction and visualization abstraction layer. The data abstraction layer acts as the database interface where all the data is been stored. The node abstraction layer collects and stores all the nodes metadata which will create relational links with the database. All the raw data that is going to be collected from the nodes will be translated into understandable engineering units at the conversion abstraction layer. Finally, the visualization abstraction layer will provide to the end user displays of the data in forms of spreadsheets and charts.

MOTE-VIEW is a passive monitor system in that it does not provide any interpretation of the displayed graphical data on behalf of the user. However, in terms of network and other failures MOTE-VIEW does not provide any self-configuration scheme.

The resource management is one of the key aspects of every wireless sensor network. Systems such as the Agent Based Power Management (Tynan et.al 2005) and SenOS (Kim & Hong 2003) have been created with that concern in mind. The Agent Based Power Management is a system that builds its architecture upon mobile agents. These intelligent entities are set responsible for local power management processing by applying energy saving strategies to the nodes of the network. This agent-based scheme is suitable for applications where the state of the network is partial visible at a known time or location. One of the major concerns of this approach is that by minimizing the transmission power, the communication range of the nodes will be reduced accordingly, threatening the network connectivity.

SenOS is managing network power resources by instructing nodes to sleep when they are not active inside the network. To achieve this, SenOS adapts a dynamic power management algorithm known as DPM. The DPM algorithm, by observing events inside the network, can generate a policy for state transitions. Based on that, all redundant nodes are placed inside a cluster with only one node awake for a period of time per cluster while the others are in a sleep mode for conserving energy. The SenOS architecture is based on a finite state machine which consists of three components. Firstly, we have the kernel which provides a state sequencer and an event queue. The second component is a transition table and the final component is a call back library.

Siphon (Wan et.al 2005) and DSN RM (Zhang et.al 2001) are two representative systems that provide traffic management functions in their architecture. Siphon, is based on a Stargate implementation of virtual sinks in order to prevent congestion at near base stations inside the network. These virtual sinks act as intermediates between the actual nodes and the base stations and they are distributed randomly inside the network. If at any point the rate of generate data increases beyond a level that exists in a predetermined threshold inside the system then the virtual sinks will redirect the traffic to other visible nodes. The visibility of the available nodes by the virtual sinks is one of the disadvantages of this approach, as there is a high probability that some nodes will be not covered by any virtual sink.

DSN RM (Distributed sensor network resource management) uses single radio nodes to evaluate each of their incoming and outgoing data rate and apply delay schemes to those nodes when necessary in order to reduce the amount of the traffic in the network. In every DSN there are a number of decision stations whose role is to act as data managers in a hierarchical format. However, the effectiveness of this technique is tightly bound on finding reliable data for every decision station inside the wireless sensor network that from its nature can provide inaccurate data during its lifetime due to connectivity and radio problems.

Table 2-1 presents a tabular evaluation of the currently available systems in terms of their organization and functionality.

Wireless Sensor Network Systems	**Reactivity**	**Architecture**	**Function**	**Energy Efficienc y**	**Adaptability**	**Memory efficiency**	**Scalability**
BOSS	Proactive	Centralised	Management System	Yes	Yes	Yes	Yes
MANNA	Proactive	Hierarchical	Management System	N/A	N/A	N/A	N/A
LEACH	Proactive	Distributed	Routing Protocol	Yes	Yes	Yes	Yes
GAF	Proactive	Distributed	Routing Protocol	Yes	Yes	Yes	Yes
WinMS	Proactive	Hierarchical	Fault Detection	Yes	Yes	Yes	Yes
Sympathy	Proactive	Centralised	Fault Detection	Yes	Yes	Yes	No
TinyDB	Passive	Centralised	Visualization Tool	Yes	No	Yes	Yes
MOTE-VIEW	Passive	Centralised	Visualization Tool	Yes	No	Yes	Yes
SensOS	Reactive	Hierarchical	Management resources	Yes	No	Yes	No
A.B.P.M	Proactive	Distributed	Management resources	Yes	Yes	Yes	No
DSN RM	Proactive	Hierarchical	Traffic Control	Yes	Yes	No	No
Siphon	Proactive	Distributed	Traffic Control	No	Yes	Yes	No

Table 2-1: Wireless sensor network systems evaluation based on designed criteria

2.3 The role of middleware in wireless sensor networks

Many researchers have identified, that a middleware inside a wireless sensor network can establish a framework for bridging the gap between applications and low levels constructs such as the physical layer of the sensor nodes (Hadim & Mohamed 2006).

A middleware can be visualized as a network managing software mechanism that will create communication bonds with the network hardware, the operating system and the actual application. A fully implemented middleware should provide to the end user a flexible interface through which actions of coordination and support will take place for multiple applications preferably in real time.

This section identifies some of the most well known middleware approaches that have been developed in recent years and classifies them according to their programming paradigm. This classification presents middlewares as virtual machines based on modular programming, virtual database systems and adaptive message oriented systems.

The use of a *virtual machine* inside a middleware is a flexible approach since it can allow a programmer to partition a large application into smaller modules. The middleware will inject and distribute those modules inside the wireless sensor network with the use of a predefined protocol that will aim to reduce the overall energy and resource consumption. The main role of the virtual machine is to interpret those distributed modules. The communication protocol can be designed based on modular programming. The use of *mobile code* can facilitate an energy efficient framework for the injection and the transmission of the application modules inside the network.

The Mate (Levis & Culler 2002) middleware is among those that use a virtual machine in order to send applications inside the wireless sensor network. The developers having identified the predominant limitations of wireless sensor networks such as energy consumption and limited bandwidth propose a new programming paradigm that is based on a tiny centric virtual machine that will allow complex programs to be very short. In order to achieve that, Mate's virtual machine acts as an abstraction layer with content specific routing.

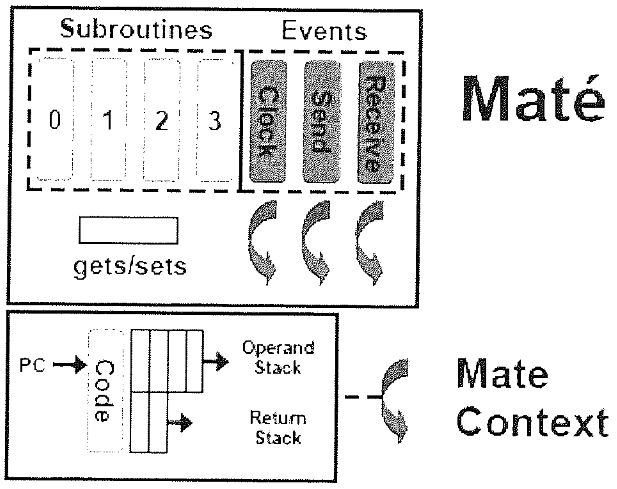


Figure 2-4: Mate architectural concept, P.Levis and D. Culler (2002)

Figure 2-4 presents Mate architecture and execution model. This high level architecture will enable the programming code to break up into small capsules of 24 instructions each that can self-replicate inside the network.

This architecture enables Mate to begin execution in response to a specific event such as a packet transmission or a time out. This is applicable through the three execution contexts that refer to an equal amount of events: clock timers, message receptions and message send requests. Each of the three contexts has an operant stack and a return address one. The operant stack will be used for instructions handling all the data while the return address stack will handle all the subroutine calls.

Every capsule that is sent inside the network includes a type and a version number. Based on that information, Mate can achieve easy version updates by adding a new number to the capsule every time a new version of the program is uploaded. However, Mate middleware suffers from the overhead that every new message introduces and also all the messages are transmitted by flooding the network in order to minimize asynchronous events notifications, raising issues with the energy consumption of every node inside the network.

Agilla (Fok et.al 2005) is a middleware that provides a mobile code paradigm for programming and making effective use of a wireless sensor network. Agilla applications consist of mobile agents that can clone or migrate across the network. The framework is based on the fact that each agent is acting as an autonomous entity inside the network allowing the developer to run parallel processes at the same time. Agilla is based on the Mate architecture in terms of the virtual machine specifications but unlike Mate, which as we described above divides an application into capsules flooding the network, Agilla uses mobile agents in order to deploy an application.

Figure 2-5 presents the Agilla model identifying the communication principle between two neighbor network nodes.

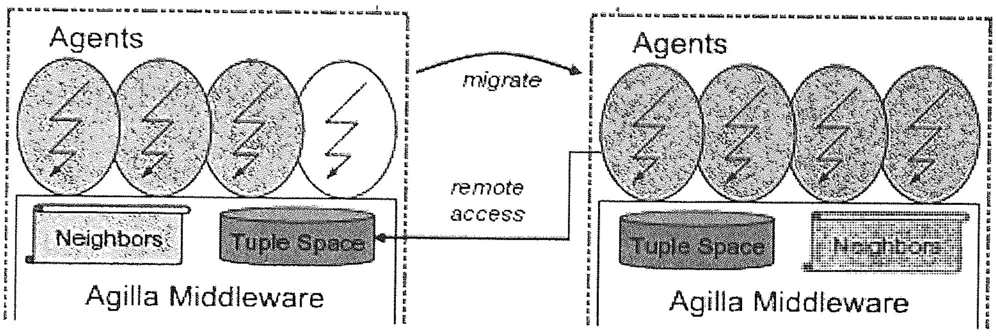


Figure 2-5: The Agilla architectural concept, C. Fok and G.Roman (2005)

In every network node we can have one or more agents residing and working independently. Their inter-communication and coordination is established by local tuple spaces that are accessible by all the agents resident in that node and a neighbor list. The local tuple space is a shared memory architecture that is addressed by field-matching. A tuple can be defined as a sequence of data objects that is inserted into the tuple space of each node by every agent. These data objects will remain in the node regardless of the agent status. In due time, an agent can retrieve an old tuple by template matching. In order to do so the sending agent must generate a query for that tuple, matching the exact same sequence of fields. The neighbor list contains the addresses of neighboring nodes and is accessible for every agent in the network that wants to clone or migrate in a different location.

Based on these attributes, Agilla allows network reprogramming thereby eliminating the power consumption cost of flooding the network. However, the lack of a hierarchical communication model for the agent society and the absence of precise real location information of every node in the network can lead to deadlocks and memory management problems.

What is known as a database middleware will visualize the whole network as a *virtual database system.* The middleware in this case will provide the user with an interface for sending queries to the sensor nodes of the system to extract the desired data.

The Cougar middleware adopts the above approach by considering the extracted sensor data to be a virtual relational database. The developers by using an SQL-like language assume that the whole network is the database. The contents of such a database are stored data and the sensor data. The stored data is represented as a virtual relationship between the sensors that participate in the network and the physical characteristics. The sensor data which is the outcome of processing functions is represented as time series that will be adapted towards the query formulation.

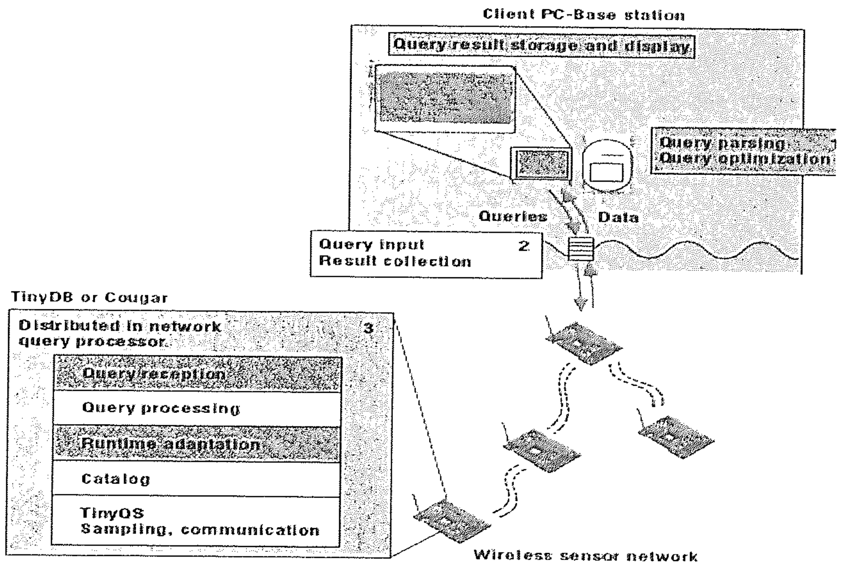


Figure 2-6: Cougar architecture for query processing, G. Gehrke and S. Madden (2002)

Figure 2-6 shows a block diagram that can explain the Cougar architecture for querying processing. One block presents the user end with the base station in the active role of transmitting and receiving queries from the wireless sensor network.

The second block presents the distributed network query processor that consists of a number of abstract data types with virtual relationships with the operating system of the network. In an SQL query format of SELECT-FROM-WHERE-GROUP-INCLUDES Cougar can allow the user to access this object relational database which mirrors the actual network.

The third class of middleware is *message oriented.* Their core architecture is based on creating a communication model that will facilitate message exchanges between nodes and the sink nodes of the wireless sensor network. To achieve that most middlewares will adapt a publish-subscribe mechanism, an asynchronous communication paradigm which allows a loose coupling between the sender and the receiver saving precious power resources.

Mires (Souto et.al 2004) middleware provides such an asynchronous communication model. This model defines three distinct faces for the nodes resident in the network. Initially the network nodes will advertise their sensed data (topic). Using a multi-hop routing algorithm Mires will route those advertisement messages to the sink node. Lastly, a user application interface will select the desired advertised topics to be monitored.

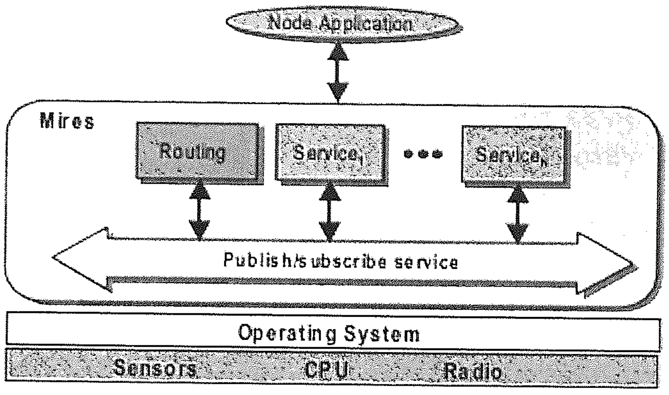


Figure 2-7: The Mires architecture, E. Souto and G. Vasconcelos (2004)

Figure 2-7 demonstrates the key characteristics of the Mires architecture. The bottom block consists of the hardware components of the node which are directly interfaced and controlled by the operating system. The middleware is placed on top of the operating system to implement its publish-subscribe communication model. This model is able to advertise the sensor data (topics) provided by the running application while it maintains a topic list provided by the node application. Mires send only messages referring to subscribed topics thus reducing like that the numbers of the transmitted packets and therefore saving energy.

2.4 Towards the design of the In-Motes middleware

In order to design and develop a successful middleware solution for a wireless sensor network that will be able to satisfy some if not all the functionalities of a network management system for monitor and control there are certain design criteria, which we described in the previous sections, and must be considered and brought forward in our design. In the following paragraphs we are going to present those design principles for a substantial middleware development.

A wireless sensor network consists of tiny devices that are battery powered and provided with a small CPU processor. Usually, and as we have already mentioned, they can be deployed in hundreds and typically in physical harsh environments. It is obvious, that after the deployment a physical contact for replacement or maintenance is highly unlikely. A middleware should be able to provide remote access to these nodes making sure that they will exhaust all their resources in terms of battery power and memory in a timely manner. Hence, one of the basic design principles for our middleware is the ability to manage limited power and resources.

Our approach in order to satisfy the above design criterion will be based in the creation of a flexible communication protocol between the nodes of our network and the base station. Thus, inspired from the GAF protocol that was described in the previous section, we are aiming to develop a protocol having node redundancy in mind, thus to regulate in an energy efficient manner which of the nodes of our wireless sensor network will be active and which ones will be in a sleep mode. Subject to our trials and the development of our middleware this protocol will be introduced both hand written in the middleware engine as well as it will be introduced as part of the running application.

Field trials such as the CodeBlue Project (Welsh et.al 2007) and the Wireless Sensing Vineyards (Holler 2008) identified that a wireless sensor network topology is subject to frequent changes due to factors such as device failure, interference, mobility and moving obstacles. Also, they proved that it is very possible that an application will grow in time, therefore mechanisms for a dynamic network topology should be available from the middleware. A middleware should be able to adapt to parameter changes caused by unexpected external factors of the environment and also provide mechanisms for fault tolerance and self configuration of the nodes inside the wireless sensor network.

Based on the above observations, and inspired from approaches similar to WinMS and Sympathy our middleware will incorporate some mechanisms for fault detection and prevention together with a flexible way of reconfiguring the network in real time. As Sympathy is a fully automated system, we will be aiming to provide some kind of automatic mechanisms in our middleware for the above design criteria without though this to be our first priority.

Unlike traditional networks, sensor networks and their applications are real-time phenomena with dynamic resources. Upon deployment of an application, core parameters such as energy usage, bandwidth and processing power cannot be predefined due to the dynamic character of these networks. A middleware should be designed with mechanisms that should allow the network to run efficiently and as long as possible. Such mechanisms include resource discovery and location awareness for the nodes in the system. Low-level programming models must be introduced as well in order to bridge the gap between the running application and the hardware.

As we mentioned before, mechanisms that are predominant in traditional networks are not sufficient to maintain the quality of service of a wireless sensor network because of constraints such as the dynamic topology and the power limitations. A middleware should be able to provide and maintain the quality of service over a long period of time while in parallel to be able to adapt in changes based on the application and on the performance metrics of the network like these of energy consumption and data delivery delay.

Inspired both form the work of the Mate and Agilla middlewares we will introduce a mechanism that will allow mobile code to be transmitted inside our network and will be able to allow changes in network parameters such as the bandwidth of each node as well as application parameters such as switching between available sensors of each node. Thus, an architecture will be developed adapting technologies such as Linda-like tuple spaces and agents/mobile code transmission with the support of a virtual machine engine.

Wireless sensor networks can be widely deployed in areas such as healthcare, rescue and military, all of these are areas where information has a certain value and is very sensitive. The environments of those areas tend to be very active and harsh, increasing the chances for malicious intrusions and attacks such as denial of service. Traditional approaches and mechanisms used to secure the network cannot be applied in this kind of network since they are heavy in terms of energy consumption. A middleware must be able to provide a secure framework for deploying applications inside the network. During the life-cycle of the network the middleware should establish protective mechanisms to ensure security requirements such as authenticity, integrity and confidentiality.

Although the above design criterion is of vast importance especially when it comes to real time applications as the ones mentioned above our approach as we will demonstrate in later chapters lacks in this area, as no security mechanisms were introduced mainly due to time limitations. Even though, this aspect of network management is not covered by our middleware we will be presenting a number of possible ways that this can be covered in the future work section of this thesis.

2.5 Summary

This chapter presents and demonstrates the wireless sensor networks as one of the predominant technologies for the 21st century. The foundational concept behind this technology is explained together with the limitations and barriers that are incorporated and need to be addressed in the process of making a wireless sensor network applicable for a number of useful applications in modern societies. The chapter opens by presenting the foundational functions of a network management system. Functions that are based upon two main attributes those of monitor and control and are critical for every wireless sensor system. A number of wireless sensor network systems are critically reviewed providing an analytical explanation of their architecture and identifying pros and cons thus helping us draw some important lessons for our proposed system. The chapter continues by presenting the middleware solution as a key player in overcoming the wireless sensor networks limitations and as our main methodology behind our proposed approach. A classification and analysis of the current middleware approaches for the wireless sensor networks is produced. The chapter concludes by presenting the role and the key design principles our middleware approach.

CHAPTER 3

The Key Methodologies of the In-Motes Middleware

3.1 Introduction

The previous chapter presented the notion of wireless sensor networks and analyzed different approaches and architectures that are aiming to overcome the limitations of these networks. In order to understand and evaluate the In-Motes middleware, the key methodologies behind it must be first identified.

Rather than re-inventing the wheel, In-Motes middleware embeds in one framework approaches that they have been established in the research community for many years. The In-Motes platform consists of concepts such as mobile code/agents, Linda-like tuplespaces, behavioral rules based on bacteria strains and federated systems, applied over the TinyOS operating system. The next sections will discuss the properties of the above methodologies that were used in our approach creating the foundation for presenting and evaluating the In-Motes middleware in later chapters.

3.2 Defining an agent/mobile code in the In-Motes environment

The limitations of centralized solutions, the need for accurate results in complex problems and the growth of decentralized approaches in the recent years are some of the reasons that agent technology has become a current trend in many areas of network technology.

Bradshaw (1997), having identified the above evolution, points out the difficulty of defining an entity as a software agent. For this reason, the author tries to distinguish the notion "agent" in two broad categories, those of ascription and description. Defining an agent entity as ascription and labeling it under one global definition is a rather difficult task as many researchers admit. "Agent is what agent does" (Bradshaw 1997:5). Table 3-1, presents various agents definitions.

System	Definition
The MuBot Agent (1994)	The term agent is used to represent two orthogonal concepts. The first is the agent ability for autonomous execution. The second is the agent's ability to perform domain oriented reasoning.
The Aima Agent (1995)	An agent is anything that can be viewed as perceiving it's environment through sensors and acting upon that environment through effectors
The Maes Agent (1995)	Autonomous agents are computational systems that inhabit some complex dynamic environment, sense and act autonomously in this environment, and by doing so realize a set of goals or tasks for which they are designed
The KidSim Agent (1994)	Let us define an agent as a persistent software entity dedicated to a specific purpose. "Persistent" distinguishes agents from subroutines; agents have their own ides about how to accomplish tasks, their own agenda.
The Hayes-Roth Agent (1995)	Intelligent agents continuously perform three functions: perception of dynamic conditions in the environment; action to affect conditions in the environment; and reasoning to interpret perceptions, solve problems, draw inferences and determine actions
The IBM Agent (1995)	Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so going, employ some knowledge or representation of the user's goals or desires
SodaBot Agent (1994)	Software agents are programs that engage in dialogues [and] negotiate and co-ordinate transfer of information.
The Brustoloni Agent (1995)	Autonomous agents are systems capable of autonomous, purposeful action in the real world

Table 3-1: Agent Definitions, Franklin & Graesser (1996)

The foundation for intelligent physical agents (FIPA) is an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies. In their specification section they provide us with what it could be characterized as the latest global definition for a mobile agent.

"An agent is a computational process that implements the autonomous, communicating functionality of an application. Agents communicate using an Agent Communication Language. An Agent is the fundamental actor of an application which combines one or more service capabilities, as published in a service description, into a unified and integrated execution model. An agent must have at least one owner, for example, based on organizational affiliation or human user ownership, and an agent must support at least one notion of identity. This notion of identity is the Agent Identifier (AID) that labels an agent so that it may be distinguished unambiguously within the Agent Universe." (FIPA, Agent Management Specification, 2005:5)

The best way to identify the agent entity as being of the description type is to view Figure 3-1, which provides us with different typologies of agents based on what those entities can do inside an environment.

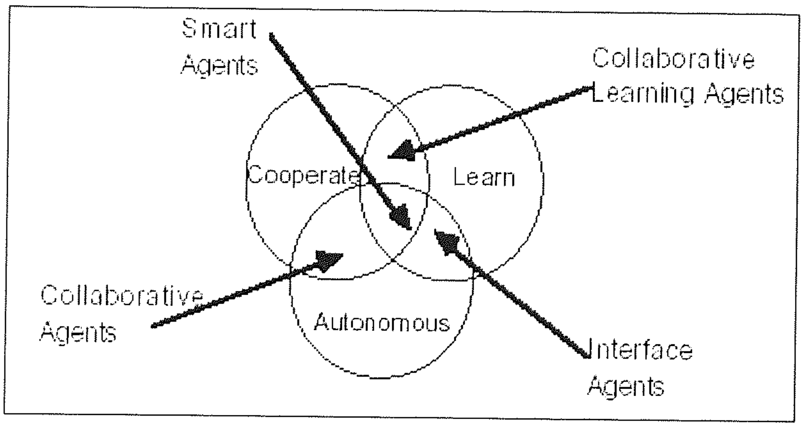


Figure 3-1: Nwana's (1996) Typology of agents

Therefore according with Nwana's (1996) primary attribute dimension we can categorize agents in the following way:

Collaborative Agents: Autonomous agents that in order to accomplish a specific task can cooperate with other agents of the society identifying a common goal.

Interface Agents: Autonomous agents that are able to learn by creating dynamic relationships with their environment.

Mobile Agents: Autonomous agents which have the ability to migrate executing "hops" in a given environment.

Internet Agents: Collaborative agents which are capable of migration inside a network.

Reactive Agents: Agents which are not proactive but can react for a collection of specific inputs.

Hybrid Agents: Those are agents that are able to combine attributes from the above categories.

Our In-Motes approach is based on mobile code that will be injected in the wireless sensor network with predefine instructions targeting the available nodes. These user predefined small fragments of code are application specific and act on behalf of the programmer inside the network at specific time intervals. Although, as we are going to see in the next chapter, they set up the communication links between the end user and the network they are not embed with any degree of social behavior neither incorporate any learning capabilities. They migration process inside the wireless sensor network is based on "hops" from one node to another according with the communication protocol we developed.

Based on the above and in our best effort to find a suitable definition for what wē will call In-Motes agents in our thesis, we conclude that an In-Motes Agent is a small computer program that is the fundamental actor of an In-Motes application which combines one or more instruction capabilities, as published in the instruction set, into a unified and integrated execution model for every node in the wireless sensor network. The In-Motes agent will have a specific agent identifier in order to be distinguished from similar agents that will co-exist locally in a node or globally inside the network at the same time. The In-Motes agents do not embed any level of learning or social capabilities, thus in the descriptive domain of a generic agent definition they pass only as individual mobile processes with pre-defined instructions that act on behalf of an end-user.

3.3 Agent architectures and communication schemes

Until now we have presented various definitions and classifications in order to understand what an agent is and how much it differs from any other software program. In this section we are going to describe how we can build an agent system that will satisfy the above theoretical views.

According to Wooldridge & Jennings (1995), agent architectures may be divided into three main categories: Deliberative, Reactive and Hybrid.

The Deliberative Architecture builds its foundation upon symbolism. More precisely a deliberative agent is going to be capable of perceiving a symbolic model of the world and through logical reasoning take some action towards a solution to a problem. The challenges of generating such architecture are located in translating and representing a real word via symbolic notation and also providing reasoning to the actions of the agents based on that notation. Two well known systems that use this kind of architecture quite successfully are IRMA (Bratman et. al 1988) and HOMER (Vere & Bickmore 1989).

The Reactive Architecture design aims to overcome the challenges that the previous architecture had introduced such as the reasoning of agents. Therefore the agents of this category don't use any kind of symbolic notation in order to take a specific action. Their mechanism is based on sensing the current environment that they are embedded in and act to specific real time calls (inputs). One key challenge of this architecture is that of what will happen if the environment changes. A system that is based on this architecture is the Pengi System by Agre & Chapman (1987). Pengi is a simulation video game where you are able to control a character.

The Hybrid architecture combines the two above architectures in one, aiming to overcome the challenges that the individual ones have. An example in this category is that of TouringMachines (Ferguson 1992). In Figure 3-2, the key components of Ferguson's (1992) architecture are demonstrated. The TouringMachine architecture provides reasoning for its agent entities based on a symbolic notation that is stored in the system under the notion of perception and is updated in every real time call. Every agent action is taking place in real time as well after the environmental inputs are produced by the three layers.

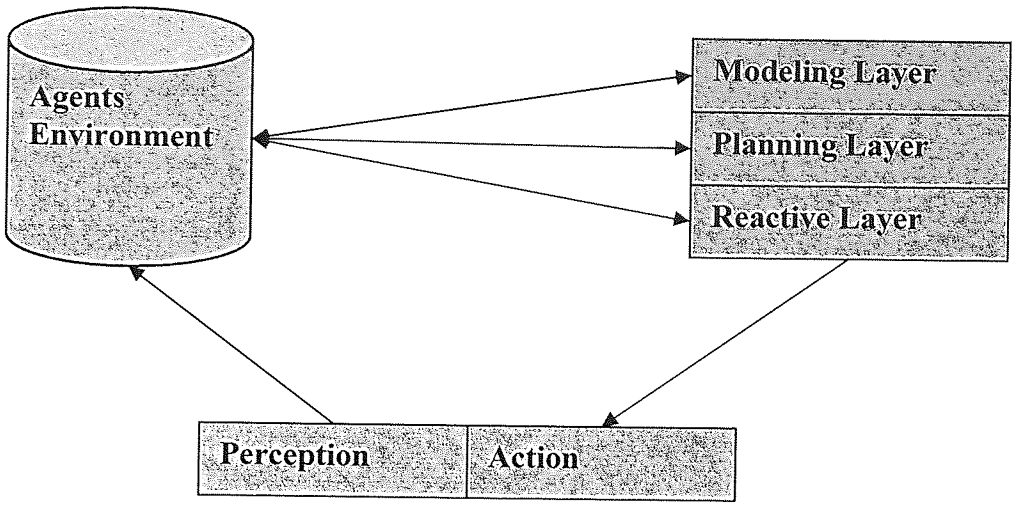


Figure 3-2: TouringMachine Architecture, Ferguson (1992)

A critical issue in any of the above architectures is communication. Designing a collection of mobile code that will be present in a network at the same time, inside our architecture the establishment of good communication bridges between all the participant software is vital. Obstacles that need to be overcome are those of deadlocks and confusion of the mobile code that is going to be active or resident in a sensor. The following table consists of a collection of proposed solutions that will allow agents to exchange information with each other, Table 3-2

Method	Description
Direct Communication	Agents communicate with each other directly using protocols such as TCP/IP
Federated Systems (Generserth & Ketchpel 1994)	Agents are organized in a federated system. Agents can communicate with each other via the use of facilitators. Overcomes the problems of direct communication cost and implementation complexity (Figure 2.3)
Broadcast Communication	Agents broadcast the message to all the agents in a given environment. One approach of doing so is the contract net one
Blackboard System(Chaib-draa & Moulin 1987)	A store where an agent is allowed to access and leave a particular message to be retrieved later by the corresponding agent. One approach to this method is the MARS (Gabri et. al) for mobile agents.

Table 3-2: Communication Schemes for Software Agents

In-Motes, is imitating the Federated Systems architecture for providing a communication scheme to its mobile code entities. We identified in the beginning of our research that agent to agent communication will generate problems with any kind of transmission in our wireless sensor network. This direct communication will increase the traffic in the form of N^2, where N is the number of agents, something that will scale badly and will eventually lead to transmission failures and packet losses. Thus we tried to find a more flexible communication scheme that will scale better inside our network.

Generserth & Ketchpel (1994) have identified that the direct communication method between agents has many benefits since it does not rely on the capabilities, existence and biases of any other subsystem or software program. However, this method is efficient only for small agent groups. For larger and more complex groups the communication cost and the implementation complexity is high, identifying the need for a more suitable communication method. Based on those facts the authors proposed the federated system method as the suitable communication architecture for complex multi-agent systems. Figure 3-3 demonstrates this approach.

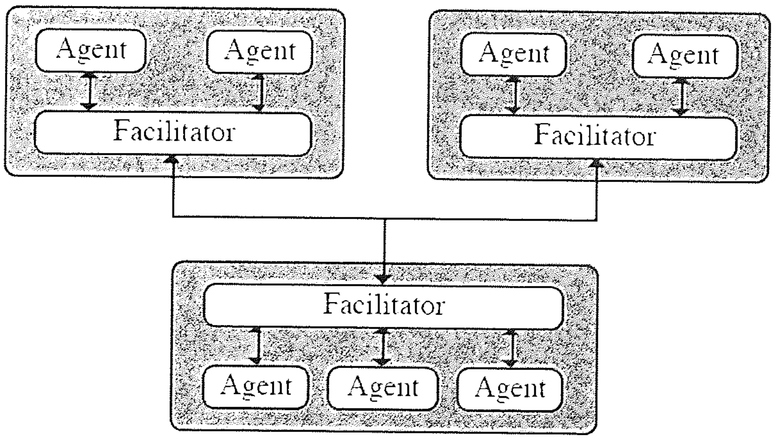


Figure 3-3: Federated System Architecture Generserth & Ketchpel (1994)

In the above figure, the federated system consists of three groups, one with three agents and two with two agents apiece. Agents they are not allowed to communicate directly with each other. Instead, they communicate only with the group agent called facilitator, and only facilitators can communicate with each other. Based on that, the traffic will scale better as it will be generated based on an Nm formula, where N is the number of facilitator agents and m is a small number of local agents which communicate with each other.

The facilitator agents will collect and transform all the data gathered by the agents and route them to the appropriate places. Thus, the agents form a "federation" in which their autonomy is passed to their facilitator agents who held responsible for fulfilling their needs.

3.4 Tuple spaces for mobile agent co-ordination

Mobile agents can produce peculiar problems when it comes to co-coordinating software components of a system (Gabri et.al 2004). A mobile agent inside a society will have a rather active life, either this is located in the co-ordination with other agent entities or with software components of the system. The deployment and the successful completion of an application may consist of many mobile agents that need to organize their actions towards a successful implementation of their primary goals and also agent entities will be distributed usually in remote sites to collect and access the services that are allocated.

Based on the above, Gabri et.al (2004) identify four main models for coordinating a mobile agent society, Figure 3-4.

		Temporal	
		Coupled	Uncoupled
Spatial	Coupled	Direct *Java Agents* *Agent-TCL*	Blackboard-Based *Ambit* *ffMain*
	Uncoupled	Meeting-Oriented *Ara* *OMG Events*	Linda-like *Jada* *TuCSoN*

Figure 3-4, Coordination Models for multi-agent systems Gabri et.al (2004)

In *direct communication*, agents establish a communication by explicitly identifying (spatial coupling) and synchronizing with the involved partners. The agents coordinate their actions by choosing a communication protocol which is usually a peer-to peer one and they are accessing all the available environmental resources in a client-server scheme (Adler 1995). Systems that have adapted this coordination model include Sumatra (Acharya et.al 1997) and Agent Tcl (Gray 1996).

However, this approach of coordination requires stable network connections something that is not, most of the time, feasible. Also, since agents have a dynamic character, any application based on them will grow, expand and even generate additional agent entities for the society in time. Thus, identifying and synchronizing at the beginning all the involved parties is not applicable and highly inflexible at an application level.

In order to overcome the limitations that the spatial coupling poses, the *meeting oriented coordination* scheme avoids all together the explicit identification of the involved parties by the agents. This is applicable by instructing the agents explicitly or implicitly to communicate and synchronize with each other at predefined meeting points. Constraints such as locality in terms that only local agents can participate in a given execution environment and assumptions such as the existence of an initiating entity for opening a meeting point are applied in order to overcome the problems of unreliability and delays. A representative system that uses this model of coordination is Ara (Peine & Stolpmann 1997). However, extreme synchronization of the agent entities is required as well as creation of additional entities that will have the role of the initiator every time a new meeting point occurs. Thus, any application in a large scale execution model will generate way too many initiators with undetermined life expectancy that could be easily susceptible to attacks by malicious agents that can take advantage of this generic time frame of the "meeting agents".

The *blackboard-based coordination* is based on shared data spaces that are local to every hosting environment. Agents can retrieve or store messages at any time as long as there is a common message identifier between the agent and the blackboard. This model fits perfectly the character of agent entities that most of the times are unpredictable in terms of scheduling a task. An agent can leave a message on the blackboard without needing to know where the corresponding receivers are and when they will read it. Also multi-tasks or queries can run simultaneously from the system simply by placing a job description on the blackboard and setting a marker that the agents will use as identifier for the job. However, they are no guarantees of how long a group of agents will take in order to accomplish a task. Also, there is a lack of inner agent coordination since each agent will work autonomously in the environment without exchanging any messages during the search and retrieval of a resource. Systems that have adapted and applied the above coordination scheme include the Ambit (Cardelli et.al 1997) and ffMAIN (Domel et.al 1997).

The *Linda-like coordination* model is based on associative blackboards called tuple spaces (Carriero & Gelernter 1989). A tuplespace is a virtual shared memory scheme that can be accessible to all the running processes of a system no matter where those are located and it consists of a collection of tuples. As in the blackboard scheme all the processes in the system don't need to know anything about other residing processes. A tuple can be defined as an ordered collection of values. For example, the tuple <18, 1980, "Dimitrios"> is a tuple of size 3, with the first two elements being an integer and the last one is a string. Tuples are accessible in the system not based on their address location rather than by their value and structure therefore it doesn't really matter where they are stored. Thus, the tuplespace can be either resident in one node or spread over several ones.

According to (Carriero & Gelernter 1989) a Linda system will have three basic primitives, out(tuple), in(template) and rd(template). The first primitive is responsible for taking a given tuple and adding it to the tuplespace. The second one returns a tuple matching the template. A template is a tuple as well but with some elements replaced by the programmer that can be matched with some of the tuples in the tuplespace. When a template matching is successful the matched tuple will be removed from the tuplespace. If no tuples matching the template that are currently in the tuplespace then the call blocks until a tuple is outputted that does match. The last of the basic primitives is identical to the in primitive with the only difference that it creates a copy of tuple rather than removing it completely when a template matching is successful.

There are two types of Linda systems, a closed and an open one. Initially all systems that were using the Linda scheme for coordinating their actions were closed, meaning the existence of a foundational requirement that was stating that all programs and processes should have been compiled at the same time along with the topology information. However, this scheme was quite inflexible especially when systems were responsible for complex and dynamic environments. Therefore, the open implementation of Linda which is currently used allows any program to connect to the system with no prior information needed by the compiler. The benefits of this type are all located in the flexibility that this scheme provides although there is an underlined threat that programs with incompatible types will try to connect simultaneously.

This associative coordination model suits well any mobile agent application. In a wide, dynamic environment that can be either the Internet or a wireless sensor network, agents will require pattern matching mechanisms to deal with issues such as the uncertainty and heterogeneity of those environments. With the Linda model those issues are embedded inside the coordination model simplifying the agent programming and reducing dramatically the application complexity.

Denti et.al (1998) proposes the addition of certain reactions inside the tuplespace scheme in order to influence the access behavior of the tuples. Thus, the tuplespace is more abstract and meaningful rather than a bag of tuples since it has its own state and can react to specific actions. Reactions by having permission to access the tuplespace can change its context and modify the semantics of the visited agents. Benefits of this approach include better control between the interaction of the agents and good integrity against malicious agents.

The In-Motes middleware is adapting the Linda open coordination model with added reactions in order to allow the specification of the In-Motes inter-agent coordination rules (Georgoulas & Blow 2006). This scheme was also chosen in order that hardware devices can communicate in a synchronized manner with the software entities. Reactions inserted in the tuplespace of a sensor, by the predefined agent, were instructing the device to act upon this input.

3.5 Network management inspired by bacterial strains

Roadknight and Marshall (2000) proposed a model for network management inspired by a community of bacterial strains. Having identified problems such as vast number of failures and the inability to accurate predict the network performance of data networks, the authors envisage a solution of a model that will use the unique way in which bacteria transfer and share genetic material, in order to propose a more robust approach for overcoming those problems.

Neural networks and genetic algorithms have been successfully implemented for overcoming problems such as the above (Roadknight et.al 1997). One of the big advantages of applying methods like these is their ability to learn from their success and failures producing an adaptive way of solving problems leading to the best fit solution.

Bacteria are simple, single cell organisms with no complicated internal structure. Their evolution is more advanced than the Darwinian "survival of the fittest" algorithm (Darwin 1859) since the individuals are capable of exchanging elements of their genetic material during their lifetime. This migration of genetic code allows a much quicker adaptability in situation where a sudden change occurs due to environmental factors (Lenski & Travisano 1994).

Their proposed management algorithm is based on the following foundational assumptions (Roadknight & Marshall 2000):

1. A number of users will be requesting services either directly from the network or through a service provider.
2. The network is established by interconnecting all the nodes.
3. Giving permission to all nodes to reach maximum capacity of servicing request often leads to poor network resource distribution.
4. Service requests cannot stay static over time.
5. New services will become available over time while old ones will become outdated.
6. The Holistic management approach is not feasible for large scale networks

Their proposed solution is identifying each node as been responsible for its own behaviour with a level of "selfishness". Decomposing a complex system into autonomous selfishly adapting subsystems has been proved to be a viable approach for overcoming complex non-linear problems (Kauffman et.al 1994).

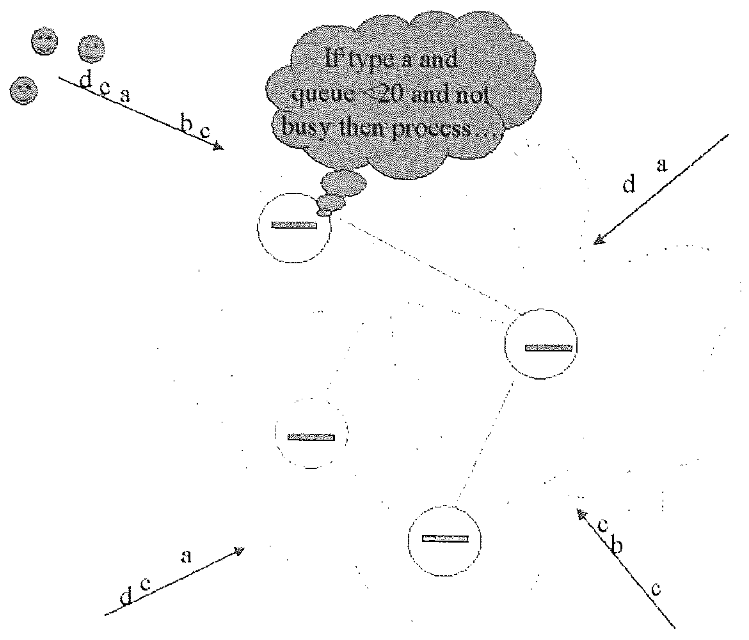


Figure 3-5, Schematic of the proposed structure Roadknight & Marshall (2000)

The Figure 3-5, demonstrates the proposed structure of the network. Their implementation supports four services, A, B, C, and D. Nodes are divided into two categories switched on (solid borders) and switched off (dashed borders) with each one of them having its own genetic material that codes for the rule set by which it lives. The rule set of each node regulates the way that each node will handle requests for a service.

The system initially populates a random selection of vertices for each active node. Each vertex is injected with a number of services that are placed in an array. If the vertex is populated by a node, the items join a queue based on the FIFO principle. If there is no node the requests are forwarded to a neighboring vertex. Every node checks if the requests are certifiable by its rule set. If a match between a rule and a request is obtained then the request is deleted and the node is rewarded. If no match is obtained the node is not rewarded and the request is forwarded. Only four requests can be processed per measuring period and the more time a node is spending to deal with a request the busier it will appear for the rest of the network community.

The authors suggest that an analogy can be created between the above rule based plasmid migration and the metabolic diversity of bacteria, capable of using various energy sources as food. Inside the network we can have the random alteration of one value into a single rule thus a process of mutation. Also, based on the level of business if a node acquires more than 4 rules through interchange, the newest rules are reserved so that no more than four rules are active thus a process of plasmid migration.

In-Motes middleware, as we are going to describe in the next chapter, is adapting this structure and rules to build a list of behavioral rules for its facilitator agents inside the wireless sensor network. Thus, we were able to generate a strongly hierarchy for our agent society where its agent is working based on what is the best for it but also for the group that it belongs too. The In-Motes behavioral rules are the following (Georgoulas & Blow 2006):

1. Each facilitator evaluates the items that arrives in its input queue on a FIFO principle
2. Only four requests can be processed in a measurement period (epoch)
3. The more time a facilitator spends processing a request, the busier it will appear to be.
4. If a facilitator's busyness is higher than 50% then the job is forwarded to the next available facilitator whose busyness is less than or equal to 50%

3.6 The TinyOS operating system

TinyOS is an open source component based operating system designed to run in wireless sensor devices. The programming core of the platform is developed to accommodate event based programming. TinyOS provides high degree of optimization by tightly coupling software and hardware components and by allowing a large number of application modules to be developed.

The execution model of the operating system is similar of that of a finite state machine but far more flexible when it comes to programming. The TinyOS event based scheduling structure enables a high level of concurrency to be dealt with by small amount of space. Each existing module continually checks the environment for specific events and responds accordingly. When an event occurs the execution context will be addressed by the module and after finishing the required processing it will be returned back to the system.

In order to deal with power consumption limitations, TinyOS instructs all modules associated with an event to declare when they finish their processing task therefore conserving CPU power. The tasks in TinyOS are programming entities that were developed to satisfy long computational modules. A TinyOS task is an execution context that runs in the background without interfering with other concurrent system events. All tasks are processed by the operating system in a FIFO scheduling algorithm.

TinyOS also permits component programming giving the freedom to an application developer to combine effectively different components under the same framework. Figure 3-6, presents a representative example of a TinyOS component whose role is to dispatch messages to a predefined location in the system. The component consists of four interrelated parts. As can be shown from the Figure 3-6 those are; a number of predefined tasks, a set of commands (upside down triangles), a block state (component frame) and a set of handlers (triangles). Components such as the above are the responsible units for all the routing, control and data transformations.

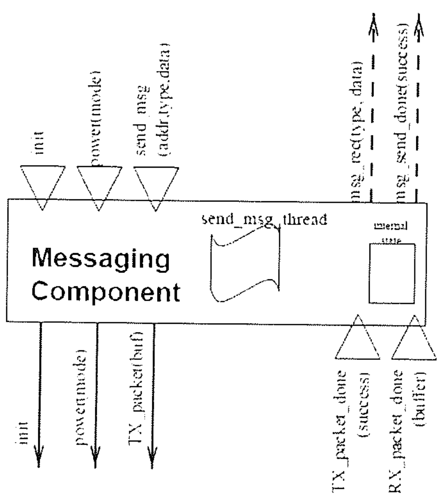


Figure 3-6, A sampling messaging component in TinyOS Hill et.al (2000)

3.7 Summary

This chapter presents the foundational theoretical concepts behind our middleware development. The notion of an agent as an ascription and description is analytically explained and a categorization of the most well known types of agents is given. A definition of what is an In-Motes agent and what characteristics this mobile process will have are presented. The predominant agent architecture schemes are explained and their communication schemes are analyzed through a series of well known systems. The coordination techniques are presented through the notion of the tuples/tuplespaces theory. Additionally, an innovative idea of network management through a parallelism of bacterial strains and concepts of biology is also incorporated. The chapter ends with an introduction to the supporting software of our middleware, namely TinyOS.

CHAPTER 4

The In-Motes Middleware Design

4.1 Introduction

The aim of this chapter is twofold. Having identified from the previous chapters the notion of a wireless sensor network, the limitations and the most feasible approaches towards a substantial system, our aim is to present a framework that would be able to satisfy those standards and if possible to overcome some of the limitations that those networks inhabit. The middleware was developed combining the axioms and techniques that were presented in Chapter 3 having in mind the development and the efficiency of the application layer for wireless sensor networks.

The In-Motes middleware can be defined as a mobile code middleware that generates a flexible framework for deploying applications in wireless sensor networks (Georgoulas & Blow 2006). In the next sections the specification and the low level design of the In-Motes middleware version 1.0 will be presented.

4.2 The specification of the In-Motes middleware

The inspiration had derived from a number of middlewares that have been developed in the recent years but mainly from those of Mate (Levis & Culler 2002) and Aggila (Fok et.al 2005). The sections of the specification part will present the In-Motes agents, the communication protocols both for the software and hardware and the coordination principles. Last but not least, the nesC language under the TinyOS environment will be recognized as our basic tool for the development of our middleware.

4.2.1 The In-Motes agents

The In-Motes agents consists of four different mobile code categories that can co-exist at the same time inside the wireless sensor network according to the user needs and the specification of the deployed application. Figure 4-1 displays the different categories.

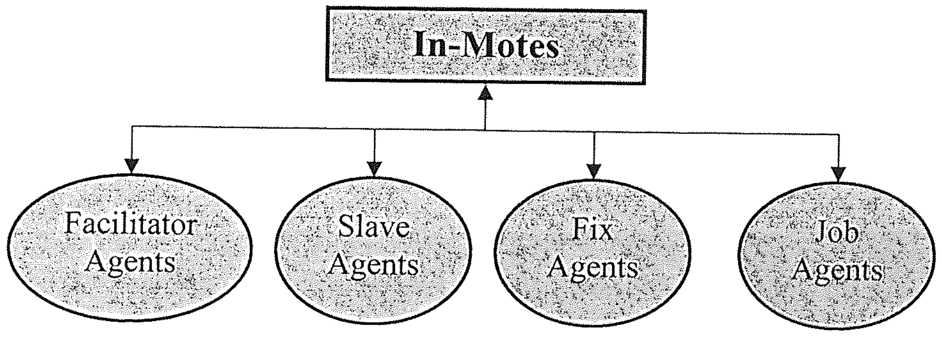


Figure 4-1: A categorization of the In-Motes Agents

The *Facilitator* category consists of In-Motes agents that are responsible for forming a "federation" of the active mobile code in the selected nodes. Their role is twofold, first they set up the communication protocol, described in the next section, and secondly they are responsible for collecting all the results from the job In-Motes agents and forwarding them to the base station for analysis. Their life expectancy in the network is tightly bound to the life expectancy of the application. They are able to exchange messages with each other but are unable to take any readings from the nodes that hosted. They have the ability to provide a simple form of decision making based on their level of business. Thus, if a facilitator level of business is above 50%

the query will be passed to the next available facilitator. Each facilitator In-Motes agent is divided into small packets of 41 bytes each, upon deployment in order to minimize message loss and deadlocks. They each consume 135 bytes of virtual memory from the In-Motes engine.

The *Slave* category consists of mobile code that is responsible for capturing the available nodes in the wireless sensor network. By the term capture we mean the ability to assign a predefined number of N nodes under the same facilitator In-Motes agent. Slave agents are practically clones of the facilitator agents and they do not provide any local decision making. After a successful capture of a node they report back to the facilitator agent and then die. Each slave agent being a facilitator clone consumes 135 bytes of virtual memory during its active period.

The *Job* category consists of mobile code that is responsible for carrying out the user requests to the wireless sensor network. Their role is to collect readings from the sensing devices of the hardware and their specification is tightly bound with the application. Thus, a job In-Motes agent could be reporting temperature, light or acceleration readings to the facilitator agent. They can report only one set of readings at the same time. Job agents are able to be transferred either by cloning or by migrating inside the wireless senor network based on the application and the available memory. Therefore, for large scale, complex applications which needed most of the memory resources, job agents are migrating while for simple applications they are cloning. The difference between cloning and migrating is based on how the code is transferred inside the wireless sensor network. Thus, a Job agent is migrating when the same code is visiting the predefined nodes alters their parameters, but it never stays resident in any of them, while with cloning, a Job agent creates multiple copies of its code that are transferred and stay resident in all the predefined nodes.

According with their status, defined as static or dynamic, In-Motes job agents are able to provide a simple level of local decision making (Georgoulas & Blow 2007). The term "static" describes In-Motes job agents which perform a single user request measurement and then die while "dynamic" describes In-Motes job agents which perform multiple measurements and respond to changes in user defined parameters. The dynamic In-Motes job agents consume 118 bytes of virtual memory and they migrate inside the system while the static ones 68 bytes and they clone inside the wireless sensor network.

The *Fix* category consists of mobile code that is used as a debugging tool for the wireless sensor network. Their role is to flush the memory of a single node in case of a problem such as buffer overflow or to flush the memory of the total number of nodes of the network. They are small in size, 25 bytes, and do not provide any local decision making.

In the next chapter the testing process for this mobile code will be presented. Through a series of simple tests we were able to inject individually the four different In-Motes agents to a number of predefined nodes and observe their functionality. It is important that the above mobile code is tested before we move on with the creation of any complex application.

4.2.2 The In-Motes architecture

The In-Motes architecture (Georgoulas & Blow 2006) is shown in Figure 4-2 and is divided in two layers. The first layer consists of the In-Motes agents that we have described above. Based on the fact that we could have one or more mobile codes active at the same time on the same node lead us to the need for a second layer that apart from the In-Motes engine would include a manager scheme for regulating issues such as context and reactions. Without this layer the In-Motes agents would have a loose hierarchy that would lead to confusion between their roles and responsibilities inside the wireless sensor network and also the system would consume unnecessary physical and virtual memory. Thus, the second layer consists of a facilitator manager, agent manager, rules manager, operation manager and an instruction manager.

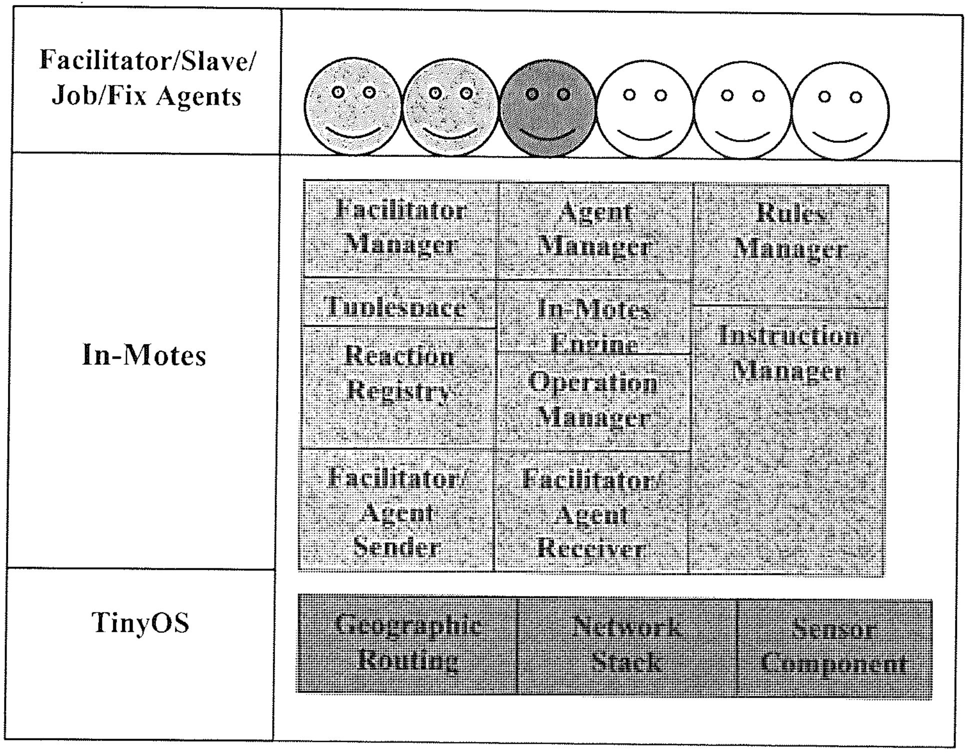


Figure 4-2: The In-Motes Architecture, the first layer consists of the In-Motes agents, the In-Motes layer sits on top of the TinyOS platform.

The *Agent Manager* is responsible for maintaining the context of the facilitator, slave and job agents. When any of those agents arrives in a mote, the agent manager allocates to the entity a specific amount of memory which is predefined by the user. With this scheme we were able to overcome the memory problem that appeared in the beginning of the development of In-Motes. Having only 4 Kb of physical memory and many active agents in the network consuming virtual memory a regulating mechanism offering an assigning function reduced problems such as stall of the mote due to insufficient system memory. We have pre-programmed the agent manager to be able to handle up to two agents due to memory restrictions. The agent manager is also responsible for notifying the In-Mote engine when an agent is ready to run in the network.

The *Rules Manager* is responsible for maintaining the behavioural rules for each facilitator. Every time a new facilitator is ready for execution it informs the instruction manager and creates a list of behavioural rules that is maintained until the termination of the facilitator agent.

The *Instruction Manager* constantly checks for new arrivals of agents in the network in order to allocate to any new agent a specific amount of instructions by retrieving the next executable instruction. Thus, agent instructions are not stored within the agent but rather are managed within In-Motes. This role is of critical importance in our middleware since TinyOS does not provide any dynamic memory allocation.

The *Facilitator Manager* is responsible for all the tuplespace and reaction operations. The facilitator manager dynamically allocates memory for every tuple insertion. Following the Mate (Levis & Culler 2002) example, each tuple can contain up to 4 fields in order to fit the 27 byte payload of a single TinyOS message. It also stores all the reactions registered by a slave or a job agent. During a tuple insertion the facilitator manager checks the registry for a match. When a match is obtained it notifies both the agent manager and the instruction manager.

The *In-Motes Engine* acts as a virtual machine on top of the TinyOS platform controlling the execution as well as all the departures and arrivals of the In-Motes agents. Initial tests of our hardware, mica2, conclude that our sensors had unreliable radio. Thus, each agent was sent into the wireless sensor network not as a single message but as numerous types of messages, Table 4-1. This provides us with a better way of controlling the agent transmissions and incase of a failure to retransmit only a number of packets rather than transmitting the whole agent again.

Type	Size (Bytes)	Content
State	20	Program counter, code size, condition code, stack pointer
Code	25	Stack pointer (points the next available instruction)
Heap	34	Five variables and their addresses
Stack	30	Five variables
Reaction	38	One reaction

Table 4-1: In-Motes messages send during deployment of an agent

Following the Mate and Agilla example the default number of instructions is 4. The engine switches context when an agent is in one of the states: sense, wait or sleep. In case of a failure such as packet loss it retransmits the packet up to R times and then stalls operation. The specific optimum value for R will depend on both the hardware and the software.

4.2.3 The In-Motes agent architecture

The In-Motes agent architecture is shown in Figure 4-3. It consists of a stack, heap and four 16 bit registers.

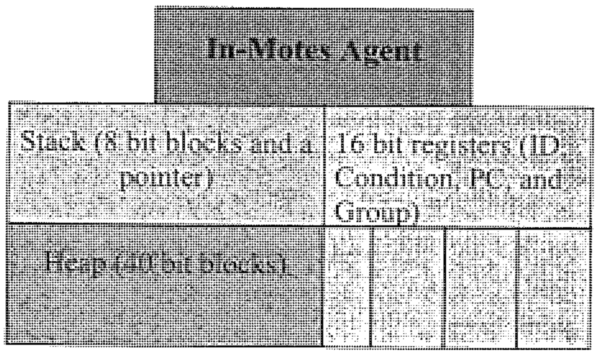

In-Motes Agent
Stack (8 bit blocks and a pointer)
16 bit registers (ID, Condition, PC, and Group)
Heap (40 bit blocks)

Figure 4-3: The In-Motes mobile agent architecture

The In-Motes agents use a predefined stack of 8 bits per block instruction. The 8 bits is the maximum instruction size an agent can use, most of the agents use 3-5 bits per instruction block. The heap consists of 40 bit blocks and is used as the store room for every In-Mote agent. The chosen number of blocks is the optimum number for handling the agent context of In-Motes. From the heap, agents can access up to 13 variables which describe the functionality and the actions of the agents. Every agent also has four 16 bit registers that contain the agent ID, the condition code, the program counter (PC) and the facilitator group ID. The agent ID which is controlled by the agent manager is chosen prior to deployment and must be unique for every agent. The slave agents that are able clone inside the network are the only ones that will be assigned with a new ID upon this action. The program counter consists of the address for the next available instruction for the given agent. It is used by the instruction manager in order to assign new instructions. The condition code is a 16 bit register that is assigned to record the execution status of each active agent in the environment. Finally, the facilitator group ID distinguishes which agents are working under the same facilitator agent and is unique for each agent "federation".

The In-Motes instruction set is based on those of Agilla and Mate. However, there are many modifications and differences in order to support the facilitator agent's scheme and the tuple space operations. A full list of the instruction set is available in Appendix A. For reference, Table 4-2 demonstrates some unique In-Motes instructions.

Instruction	**Opcode**	**Parameters**	**Return Val.**	**Description**
getslv	0X21	value	id	Get a slaves id
sclone	0X2d	location	n/a	Clone slave
fwait	0X1b	n/a	n/a	Stops slaves execution allows it to wait for a reaction
fout	0X34	tuple	n/a	Inserts a tuple to facilitator local tuple space
srd	0X38	template	tuple	Block slave to find a tuple

Table 4-2: An example of unique In-Motes agent instructions

The In-Motes agent instruction set can be divided into three categories: General purpose, tuple and facilitator instructions. The *general purpose instructions* are identical to those of Agilla and Mate such as pushc, halt, sense, and putled. However, the In-Motes instruction set includes instructions such as srd, waket and sleept that enable agents to act with better hierarchy and with more robustness. For example, an In-Motes dynamic job agent can perform some application specific actions providing reasoning with the instructions waket and sleept, something that is not supported by the Agilla agents, and it would take the flooding of the whole network with timer capsules in Mate in order to achieve something equivalent.

The *tuple instructions* allow an In-Motes agent to communicate with the hardware through the tuple space that each sensor hosts. Instructions such as out and rd are common for both In-Motes and Agilla agents and for every host they are providing with the accessibility of the local tuple space. Additional instructions such as fout and srd were added in order that federation communication scheme be applicable and that facilitator and slave tuples can be accessed both remotely and locally from the In-Motes agents.

The *facilitator instructions* allow an In-Motes agent to move or clone from one node to another following the federation scheme. Those instructions are unique for the In-Motes agents and they make our communication protocol, which we will present in the next section, applicable to the In-Motes agents. Instructions such as getslv and sclone are applicable only to facilitator and slave agents and define the character of them inside the wireless sensor network. Lastly, instructions such as fbusy and freqst are derived from the behavioural rules that each facilitator agent must comply with.

4.2.4 The communication protocol

The In-Motes communication protocol is based on the federation communication scheme which was described in chapter 3. In order to implement it, we are sending a facilitator In-Motes agent to the network in order to capture and create facilitator and slave nodes before any user requests or the actual application is forwarded. The life cycle of a facilitator In-Motes agent is shown in Figure 4-4 (Georgoulas & Blow 2006).

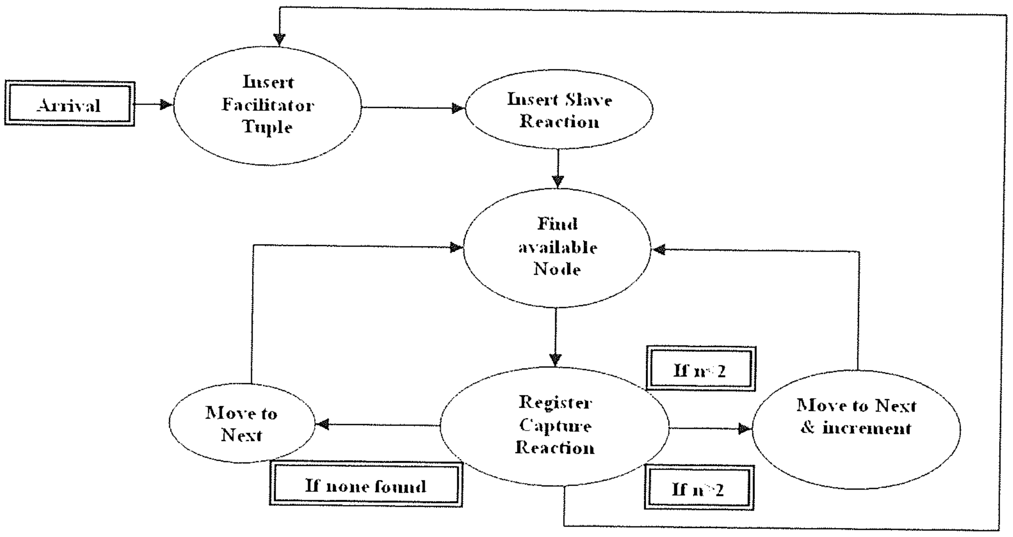


Figure 4-4: The life cycle of the In-Motes facilitator agent

The facilitator agent works by continuously checking whether any of the nodes are available for capture. The user sends a single facilitator into the wireless sensor network, although this is not limited by our infrastructure, allowing more than one facilitator to be deployed in large scale applications where the nodes exceed the total number of 20. Preliminary tests that we run and we will demonstrate in the next chapter conclude that for a wireless network that consists of more than 10 nodes a parallel approach where two or more facilitator agents deployed at the same time is more applicable as it consumes less virtual memory and establishes the communication protocol faster.

Upon arrival at the first available node, the facilitator will insert a facilitator tuple into the tuple space assigning thus the first facilitator node. The capturing procedure takes place when a facilitator agent during its migration registers a capture or a slave reaction to the corresponding node. An alternative is for the facilitator agent to clone rather than migrate and generate what we have described earlier as a slave agent inside the wireless network. During our trials both approaches were tested equally successfully.

A counter will be incremented every time a capture reaction takes place; here we restrict the registration of two agents under each facilitator purely based on the motes we had available. When the counter reaches two, the facilitator agent will migrate again to the next available node assigning this time around a new facilitator tuple and slave reaction and the capturing procedure will repeat.

It is expected that during the lifetime of a wireless sensor network some nodes will eventually die and information will be lost. In-Motes can adapt and dynamically take actions upon unexpected scenarios like the ones mentioned above. If a facilitator node goes down the network will dynamically adapt since the lifecycle of the facilitator agent that we described above never terminates and a new capturing procedure will take place.

4.2.5 The nesC programming language

The In-Motes middleware architecture was developed using the nesC programming language which is an extension of the C language. The nesC programming model is suitable for networked embedded systems since in its programming model it incorporates elements such as event driven execution and component oriented application design (Gay et.al 2001). Also nesC has been used to implement the operating system we were using, TinyOS. Thus, it was easier to create and generate the communication paths in a language that the operating system was built in.

Our middleware with the use of the TinyOS operating system handling the sensor devices addresses a number of challenges upon development. As presented above, the agents and the environment we are planning to generate and deploy are highly event driven. The motes as well as the agents of our middleware are reacting to the changes in the environment and user inputs and also they are working under a concurrent framework which is trying to eliminate network problems such as race conditions between messages or sensor acquisitions. The nesC programming language provides a very rich concurrency model with a powerful compiler. Moreover, its programming model is suitable for event driven environments and devices since it is based on a number of bidirectional interfaces.

The second challenge faced during the creation of In-Motes was that of limited resources. We have already made reference in previous chapters to the fact that the mote life expectancy is tightly bound with the life expectancy of the batteries. NesC is suitable for developing applications with small need in power resources since its model is suitable for reducing code size and its compiler can perform dead-code elimination. Also the concurrent behavior that it adapts needs very limited resources in order to be executed.

Thirdly, with the creation of In-Motes applications, we needed to address the issue of reliability in a real environment. For In-Motes applications such as environmental monitoring, that we will present in the next chapter, it was necessary to collect data in a monthly period reducing as much as possible the human intervention. Thus, we needed a system that would be able to provide the smallest number of run time errors possible. Once again the nesC programming language was able to offer component instantiation and extensive compile- time analysis thus creating programming modules that are able to detect and eliminate errors such as the one described above.

Lastly, the developers of the nesC language, in order to create a simple programming model that will support and reflect to the TinyOS design, made it as a static language although it is not restricted to that character (Gay et.al 2001). This means that the TinyOS operating system does not offer dynamic memory allocation for its applications although the nesC programming core can support it. Even though this was a limitation for our middleware development, which as we said earlier was finding insufficient the 4 Kb of motes physical memory, with the use of nesC programming model we were able to develop a dynamic virtual memory allocation scheme for our middleware and more specific for the In-Motes agents.

4.2.6 The interface programming language

The In-Motes middleware interface has been implemented using Java. The reasons behind this selection can be found in the object oriented character of the language, the Java virtual machine (JVM) and the remote method invocation (RMI) which is applicable for remote objects.

Java is an object oriented language and much of its syntax derives from C and C ++ programming languages but with a simpler object model and fewer low-level facilities. Since In-Motes is a multi-agent platform applied in a distributed environment, Java was the ideal choice as it is suitable for distributed programming. To that extent, the fact that nesC, which is a modular version of C, was the unifying language between our middleware and the operating system made Java the most suitable language for combining distributed nesC objects, used at the low level architecture with query oriented messages, used from the user front end.

Using the remote method invocation that Java provides, we were able to invoke methods on objects residing on other machines, in our case sensors, as well as invoking methods on local objects. Figure 4-5 demonstrates how this is applicable in In-Motes middleware. A local java object hosted by the application which was usually the query of the user could be easily accessible to the remote java object of the sensor, the resident agent. Also, based on the communication protocol we described earlier a local java object could invoke the methods of a nesC object residing in a different sensor, with the use of the Java remote method invocation.

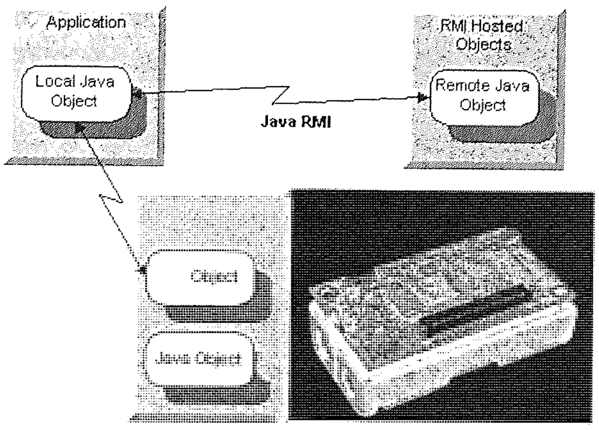


Figure 4-5: Representation of the Java remote method invocation

The concept of the Java virtual machine states that programs written in Java are compiled into a machine language, commonly referred to as Java byte code. This byte code is accessible to all machines as long as they have an interpreter installed,

The use of the same byte code for all platforms allows Java to be described as "compile once, run anywhere", as opposed to "write once, compile anywhere", which describes cross-platform compiled languages such as Ada and Pascal. This description fitted perfectly all the applications we were sending in the wireless sensor network as well as the actual middleware since both were compiled once during the lifetime of the network. The Java virtual machine also enables such unique features as automated exception handling which provides 'root-cause' debugging information for every software error (exception) independent of the source code. This attribute helped with fixing problems that were caused in the wireless sensor network by the individual sensors such as when a sensor was stalling its operation.

In order to create a flexible and easy to use interface for our middleware we used the Swing toolkit for Java. Swing provided us with a powerful application programming interface (API) that enabled us to create the graphical user interface

(GUI) for In-Motes. Swing GUI Components includes everything from buttons to split panes to tables. Many components are capable also of sorting, printing, and drag and drop. In-Motes uses the Java standard edition provided by Sun Microsystems, Figure 4-7.

Figure 4-7: Java SE overview taken from http://www.java.com [03/02/08]

4.3 The Design of the In-Motes middleware

The sections of the low level design will cover the functional modules of our middleware recognizing how Java modules together with the nesC programming language had constructed the high level specification which was described above. Moving from the theoretical perspectives to the practical ones we provide a look behind the scenes of our middleware by explaining how issues such as In-Motes agents, tuple spaces and communication protocol were effectively created.

The In-Motes middleware consists of two packages, one containing the Java code for the front end of the middleware and the other containing the nesC modules, Figure 4-8.

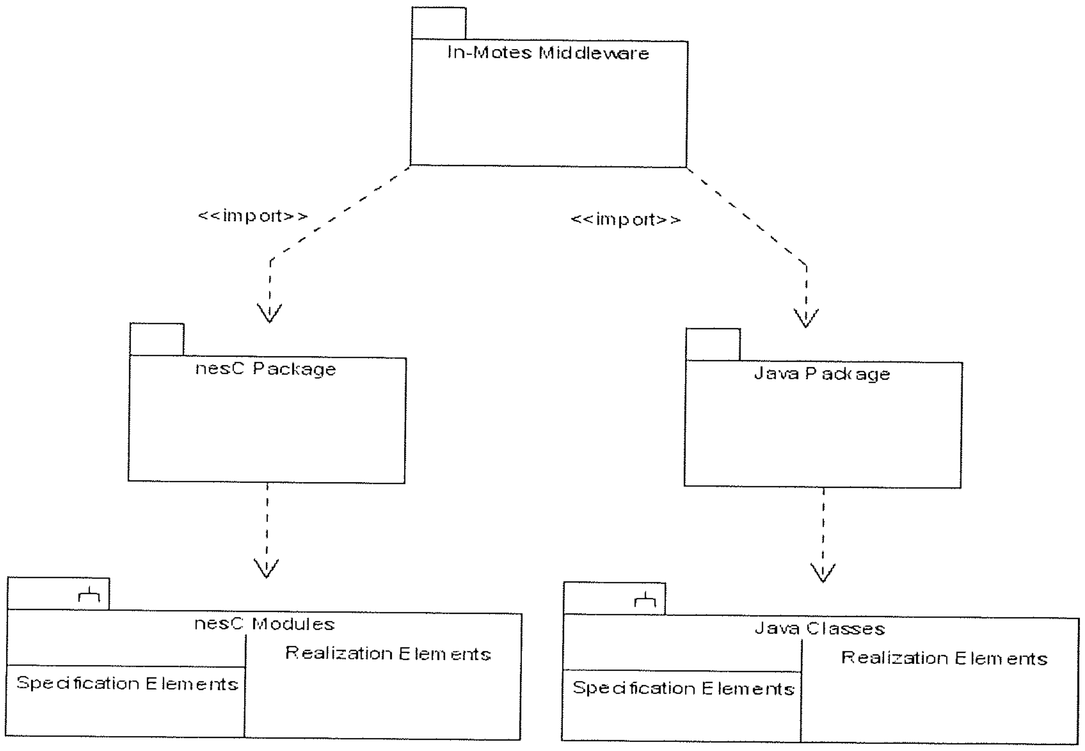


Figure 4-8: The In-Motes Middleware packages

4.3.1 The In-Motes Java Package

The In-Motes middleware Java package consists of 26 main Java classes and 4 individual sub packages with the corresponding classes of each. Each one of the main classes imports various methods from the sub packages such as the variables and messages which contain methods responsible for the interface message parsing and for the sensor variables, Figure 4-9. The next paragraphs briefly discuss some of the most important Java classes and methods of the In-Motes middleware.

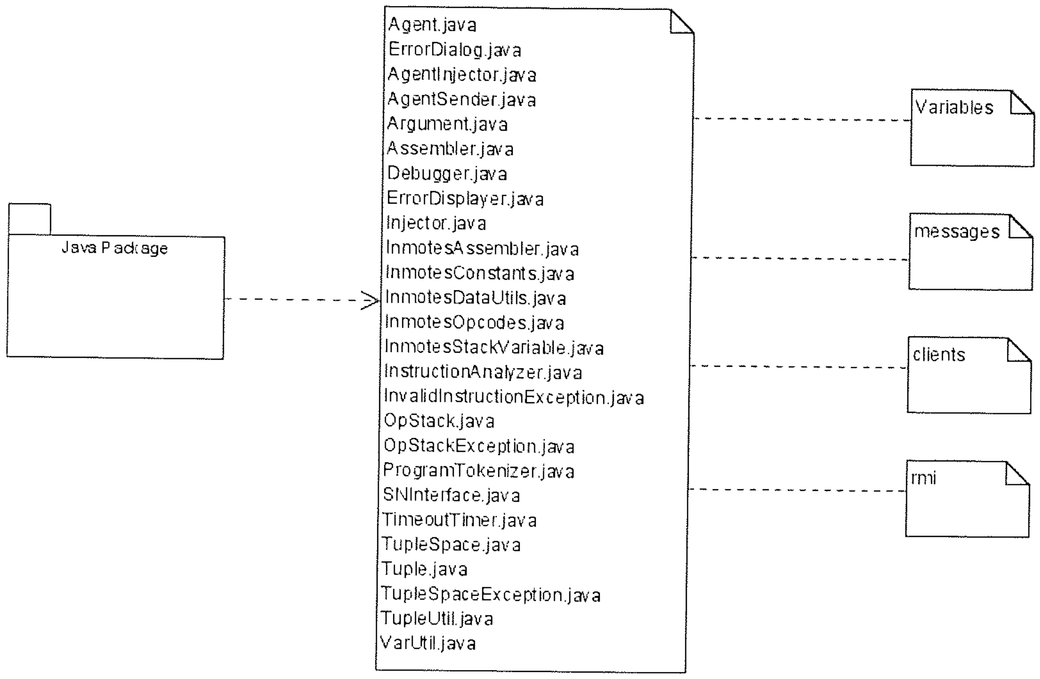


Figure 4-9: The In-Motes Middleware Java package decomposition

Agent.java

This class is one of the most important in the whole system as it contains the agent descriptions. It imports methods from the variable sub package and implements InmotesConstants and InmotesOpcodes. The class method below is responsible for outputting the agent ID, the condition code, the program counter (PC) and the facilitator group ID for every active agent in the system, Figure 4-10.

```
public Agent(InmotesAgentID id, int pc, int codeSize, int condition)
{
            this.id = id;
            this.pc = pc;
            this.codeSize = codeSize;
            this.condition = condition;
            code = new byte[codeSize];
            for (int i = 0; i < INMOTES_HEAP_SIZE; i++) {
                  heap[i] = new InmotesInvalidVariable();
            }
```

Figure 4-10: The class method Agent containing some of the agent descriptions

Also the class allows the agents to move to the next available instruction or halt their operation according to their instruction set.

AgentSender.java

This class is responsible for all the messages that will be exchanged between the agents and between the user and the agents. It imports methods both from the variables and messages sub package. The methods of the class allow the encapsulation of an agent context and provide the destination it wants to migrate to, Figure 4-11.

```
public void send(InmotesLocation dest, Agent agent) {
            Debugger.dbg("AgentSender", "Sending " + agent);
            queue.add(new PendingAgent(dest, agent));
            synchronized(queue) {
                  try {
                        queue.notifyAll();
                  } catch(IllegalMonitorStateException e) {
                        e.printStackTrace();
                  }
            }
      }
```

Figure 4-11: This void method regulates the transmission of the sending agent

Also the void method of the class specifies that the sender thread sits in a loop where sending agents are placed into a queue.

InmotesConstants.java

In this class we have stored all the constants the In-Motes middleware is using such as the maximum number in tuples, the size of the tuples and the maximum number of agents a node can host per measuring period (epoch), Figure 4-12. Also we have stored all the active message types that the agents will exchange during the lifetime in the wireless sensor network as well as the maximum number of user requests a facilitator can handle before passing the job description to the next available facilitator (level of busyness).

```
// Agent context specification
public static int INMOTES_OPDEPTH = 20;// depth of operand stack
public static int INMOTES_TS_SIZE = 100;// size of tuple space stored
                                        //in data memory
public static int INMOTES_NUM_AGENTS = 2;// max no of agent on a node
public static int INMOTES_HEAP_SIZE = 12;// size of heap on each
                                         //agent
public static int INMOTES_CODE_BLOCK_SIZE = 22; // the size of each
                                                //code block
public static int INMOTES_NUM_CODE_BLOCKS = 20; // the total number
                                                //of code blocks
public static int INMOTES_SENSOR_PERIOD = 2048; // time between
                                    //taking sensor measurements
public static int INMOTES_MSG_TIMEOUT = 512;
public static int INMOTES_MSG_RETRY = 256;

// Active message types
      static final int AM_INMOTESSTATEMSG      = 0x10; // 16
      static final int AM_INMOTESCODEMSG       = 0x11; // 17
      static final int AM_INMOTESHEAPMSG       = 0x12; // 18
      static final int AM_INMOTESOPSTACKMSG    = 0x13; // 19
      static final int AM_INMOTESRXNMSG        = 0x14; // 20

      static final int AM_INMOTESACKSTATEMSG     = 0x15; // 21
      static final int AM_INMOTESACKCODEMSG      = 0x16; // 22
      static final int AM_INMOTESACKHEAPMSG      = 0x17; // 23
      static final int AM_INMOTESACKOPSTACKMSG   = 0x18; // 24
      //... some code is missing
```

Figure 4-12: The agent context specification of In-Motes

The class methods as we have seen above are invoked both from the Agent class as well as from the AgentSender one. This class was also frequently modified since it contained the agent context specification. For different applications we were using different values. The above ones are the optimum ones that generate the less memory problems for an average size application such as the In-Motes Bins one which we will describe in the next chapter.

InmotesOpcodes.java

In a similar way of the above interface class we have created the InmotesOpcodes class. In this class we stored all the instructions that were available in the system. Zero operand, one operand two and three operand instructions were stored and they were accessible from the AgentSender class, Figure 4-13.

```
// zero operand instructions
public static final byte OPfhalt      = 0x00;
public static final byte OPfloc       = 0x01;
public static final byte OPfaid       = 0x02;
public static final byte OPrand       = 0x03;
public static final byte OPfcpush     = 0x04;
public static final byte OPfdepth     = 0x05;
public static final byte OPswait      = 0x0a;

// One operand instructions
public static final byte OPsmove      = 0x17;
public static final byte OPfmove      = 0x18;
public static final byte OPsclone     = 0x19;
public static final byte OPfclone     = 0x1a;
public static final byte OPfgetvars   = 0x1b;
public static final byte OPsetvars    = 0x1c;
public static final byte OPfgetnbr    = 0x1d;
public static final byte OPcisnbr     = 0x1e;
public static final byte OPssense     = 0x1f;

// Two and three operand-instructions
public static final byte OPdist       = 0x21;
public static final byte OPswap       = 0x22;
public static final byte OPland       = 0x23;
public static final byte OPlor        = 0x24;
public static final byte OPand        = 0x25;
```

Figure 4-13: Part of the In-Motes Instruction Set

Tuple.java

This class was defining and generating the tuples of our system. The methods that exist in this class were allowing the retrieval of the number of bytes contained within the field of the tuple and template matching when a specific tuple was matching a template, Figure 4-14.

```
public boolean matches(Tuple t) {
            boolean result = true;
            if (t.size() == size()) {
                  for (int i = 0; i < size() && result; i++) {
                        System.out.println("Tuple.matches(): " + i +
" Comparing: " + getField(i) + ", with " + t.getField(i));
                        result = getField(i).matches(t.getField(i));
                  }
            } else result = false;
            return result;
      }
```

Figure 4-14: Method that is accessed for tuple matching

The class was importing methods from the variables sub package and the java.util package.

<u>**Tuplespace.java**</u>

This class through its methods provides all the tuple space operations for the middleware. The most important methods of the class are the ones that produce what we described in earlier sections as template matching, Figure 4-15.

```
public Tuple in(Tuple template) {
            Debugger.dbg("TupleSpace.in", "||----> Performing an in()
operation.");
            Tuple result = inp(template);
            if (result != null) {
                  return result;
            } else {
                  while(true) {
                        synchronized(blocked) {
                              try {
                                    Debugger.dbg("||---->
Tuplespace.in", "BLOCKED while doing an in().");
                                    blocked.wait();
                                    Debugger.dbg("||---->
Tuplespace.in", "UNBLOCKED while doing an in().");
                              } catch(Exception e) {
                                    e.printStackTrace();
                              }
                              Debugger.dbg("||----> Tuplespace.in",
"Trying to find a match.");
                              result = inp(template);
                              if (result != null)
                                    return result;
```

Figure 4-15: Template matching method returns Returns a tuple matching for the specified template

The above method Returns a tuple matching the specified template. If no match is found then it blocks until there is an available one. If more than one match is found, it chooses and returns one randomly. The tuple is kept in the tuple space. Also this class provides a method that allows agents to flush their tuple. This action was allowing a developer to continue reusing the same agent rather than resending a new one every time a user request was satisfied. This method was invoked and used by all the facilitator agents of our wireless sensor network, Figure 4-16.

```
public void flush(AgentID aID) {
          synchronized(ts) {
               for (int i = 0; i < ts.size(); i++) {
               Tuple t = (Tuple)ts.get(i);
                    if (t.isOwnedBy(aID)) {
                         ts.remove(i);
                         i--;
```

Figure 4-16: Method used by the facilitator agent for flushing tuples

<u>Debugger, ErrorDialog, OpStackException, TupleSpaceException InvalidInstructionException {.java}</u>

We have mentioned in previous sections that one of the main problems we were facing during the development of the middleware was that the sensors we were using were provided with built in specification. That implied that errors that were produced during any kind of testing of the system were difficult to identify. Therefore In-Motes comes with a built in debugging capabilities that are generated from the above classes. These classes were able to produce warning messages or if the debugger option was selected to output a number of variables to the end user allowing him to monitor better both the behavior of the deployed application as well as the correct functionality of the middleware.

AgentInjector.java

The AgentInjector class contains the Graphical User Interface (GUI) for the In-Motes middleware. It is the basic class of the system that also contains some of the main methods to be executed firstly. In particular, it imports a number of Java SE packages such as the swing and the java.awt.event as well as TinyOS packages such as net.tinyos.message and net.tinyos.packet.

The clients sub package

The clients sub package, Figure 4-17, contains the oscilloscope application which was embedded in our middleware in order for the user to have a graphical display of the readings of every sensor in real time.

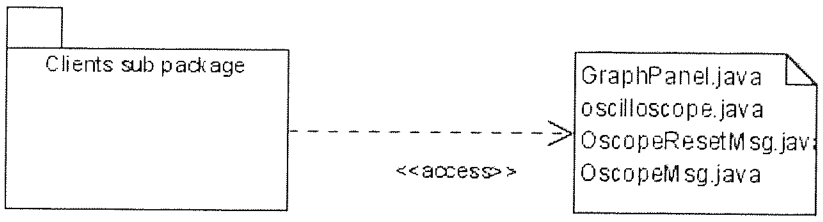


Figure 4-17: The Clients sub package

The source code of the Java classes was obtained from the TinyOS open source package where the oscilloscope application is available. Minor modifications were made in order to embed this application in our middleware.

The messages and variables sub package

The messages and variable sub packages contain classes with methods that are invoked from all the main classes of the middleware. The first package contains all the message types that are exchanged in our system such as the acknowledgement messages and the state messages. The variable one is a collection of classes with methods that are generating the various different types and ids for the agents and the sensors of the system as can be seen from the following method, Figure 4-18.

```
private String type2String(int dType) {
            switch(dType) {
                  case INMOTES_TYPE_INVALID:
                        return "INVALID";
                  case INMOTES_TYPE_VALUE:
                        return "VALUE";
                  case INMOTES_TYPE_READING:
                        return "READING";
                  case INMOTES_TYPE_STRING:
                        return "STRING";
                  case INMOTES_TYPE_ANY:
                        return "ANY";
                  case INMOTES_TYPE_TYPE:
                        return "TYPE";
                  case INMOTES_TYPE_AGENTID:
                        return "AGENTID";
                  case INMOTES_TYPE_LOCATION:
                        return "LOCATION";
                  default:
                        return "UNKNOWN";
            }
      }
```

Figure 4-18: The different In-Motes sensor types parsed as strings

The rmi sub package

This package contains 4 classes all together responsible for the remote actions that are available in our system and more precisely with remote agents and tuple access. For example, the AgentInjectorRMI.java class specifies the interface of the object that allows a remote host to inject an agent into our wireless sensor network.

```
package edu.ee.acnr.inmotes.rmi.agentInjector;

import java.rmi.Remote;
import java.rmi.RemoteException;
import edu.ee.acnr.inmotes.*;
import edu.ee.acnr.inmotes.variables.*;

public interface AgentInjectorRMI extends Remote
{
      public boolean inject(InmotesLocation dest, Agent ma) throws
RemoteException;
}
```

Figure 4-19: The In-Motes remote agent method invocation

4.3.2 The In-Motes nesC Package

The In-Motes middleware nesC package consists of 4 sub packages that contain all the important modules in order for the middleware to communicate both with the TinyOS operating system as well as with the Java graphical user interface (GUI), Figure 4-20. Those modules as we are going to see in the next paragraphs generate a concurrent structure and define system parameters such as the message interfaces that are responsible for transferring the obtained data from the sensors to the user and the In-Motes engine components.

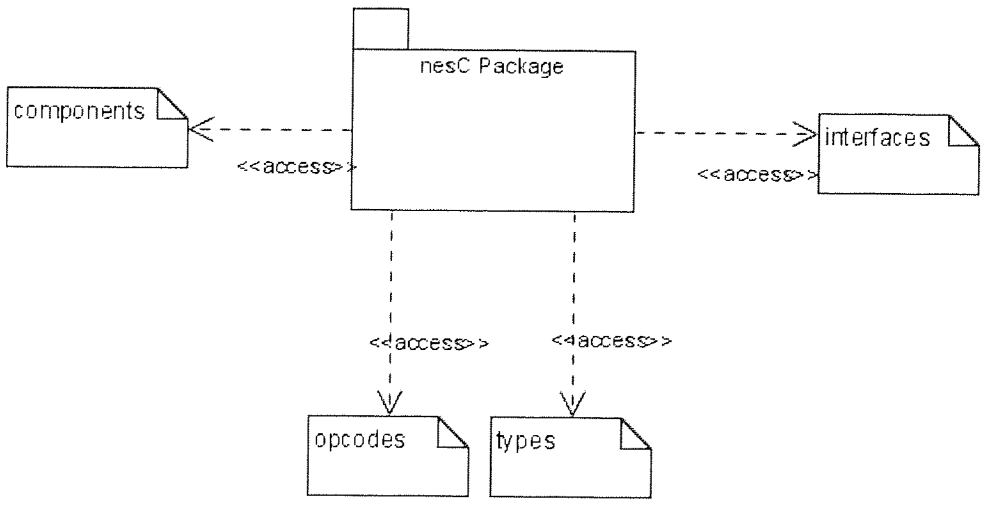


Figure 4-20: The nesC package decomposition

The types sub package

The types sub package contains 3 nesC modules that initialize and set the parameters for the In-Motes engine. Namely, Inmotes.h sets the network size, the communication of the base station with the nodes of the wireless sensor network and also all the middleware message definitions. Also, it specifies amongst others the size of the operand stack in bytes, the maximum number of agents on a node, size of heap on each agent and the total number of code blocks, Figure 4-21.

```
enum {
INMOTES_OPSTACK_SIZE       = 110
INMOTES_SLAVE_NUM_AGENTS = 4
INMOTES_FAC_NUM_AGENTS     = 2
INMOTES_NUM_AGENTS         = 2
INMOTES_HEAP_SIZE          = 12
INMOTES_CODE_BLOCK_SIZE   = 22
INMOTES_NUM_CODE_BLOCKS   = 15
} InmotesSizeConstants;
```

Figure 4-21: Some of the In-Motes engine parameters

The above values where optimum and depended upon the application and the size of the wireless sensor network. For example, the number of agents was set to 4 as that was the maximum number of concurrent agents that would be active under the same facilitator for our applications.

The Tuplespace.h module initializes all the optimum values for the tuplespaces of the middleware. Namely, the size of RAM for every tuple space in bytes, the maximum number of field bytes in a tuple, the maximum number of times a request is sent and the maximum number of reactions. Also, it generates all the tuplespace operation request/response messages as can be seen below, Figure 4-22.

```
typedef struct InmotesTSReqMsg {
InmotesLocation destLoc;
InmotesLocation replyLoc;
uint8_t op;
InmotesTuple tuple; // agent template
} InmotesTSReqMsg; // 36 bytes
typedef struct InmotesTSResMsg {
InmotesLocation destLoc;
uint8_t op;
bool success;
InmotesTuple tuple;
} InmotesTSResMsg;
```

Figure 4-22: Request/Response tuplespace messages

The MigrationMsgs.h module generates and initializes all the message parameters that will be exchanged between the agents and between the agents and the In-Motes engine. Thus, the module produces the optimum values for the maximum time a node will be waiting for an acknowledgement, the maximum number of agents sent to a node and the maximum number of agents that had been received from a node. Also, as can been seen below it includes all the agents migration message contents and all the agent migration states, Figure 4-23.

```
typedef enum {
  INMOTES_RECEIVED_NOTHING   = 0,
  INMOTES_RECEIVED_CODE      = 1,
  INMOTES_RECEIVED_STATE     = 2,
  INMOTES_RECEIVED_OPSTACK   = 4,
  INMOTES_RECEIVED_HEAP      = 8,
  INMOTES_RECEIVED_RXN       = 16,
  INMOTES_AGENT_READY = INMOTES_RECEIVED_CODE |INMOTES_RECEIVED_STATE
| INMOTES_RECEIVED_OPSTACK|INMOTES_RECEIVED_HEAP|INMOTES_RECEIVED_RXN
```

Figure 4-23: The In-Motes agent migration messages

The interfaces sub package

The interfaces sub package contains a number of modules that handle mainly the execution model of the In-Motes middlewares. Their modules specify the gateways of the hardware components to the java graphical user interface. For example AgentExecutorI.nc provides the interface for executable agents of the wireless sensor network and checks if the middleware is in an error state. The below code is taken from the BehaviorRulesI.nc which produces the interface for the list of behavioral rules to be generated for every facilitator agent, Figure 4-24.

```
includes Inmotes;

/**
 * This interface is provided by the component that can produce the
behavioral rules list
 */

interface BehaviorExecutorI {
  /**
   * Produces the list.
   *
   */
  command result_t run(BehaviorRulesContext* agent);

  /**
   * Returns SUCCESS if a facilitator agent can satisfy the rules
list
   * else returns FAIL.
   */
  command result_t inErrorState();
```

Figure 4-24: The nesC interface for producing the behavioral rules list

The components sub package

The components sub package contains all the structural modules of the In-Motes middleware. In total there are 31 different modules that combine all the various managers of the system that we presented above such as the Facilitator manager and the Rules manager. Thus, the modules specification identifies amongst others, how an agent can handle its context, how the middleware is orchestrating all of the components involved with sending an agent to a remote node, the In-Motes tuple space implementation with the use of the mote's internal data memory and the In-Motes engine.

The opcodes sub package

The last and largest in terms of modules sub package of the nesC is the opcodes one. Inside there is a collection of modules that handle every single Opcode of the In-Motes middleware. The system individual actions such as whether the type of the second variable on the stack matches the TYPE variable on top of the stack and every single instruction that In-Motes is using are generated inside 103 different opcodes modules.

4.4 Summary

This chapter presents the development of the In-Motes middleware. Initially we present the specification of the middleware by identifying key concepts such as the In-Motes agents and the architecture both for the middleware and the agents of our system. The communication protocol is also explained identifying how the facilitator scheme was applied in our framework. The selection of the two programming languages that In-Motes is using, Java and nesC, are substantiated and reasoned having always as reference both the middleware and the nature of the network it was created for. Moving one step further we present the two packages that generate the In-Motes middleware describing through modules and classes the creation and design of some of the theoretical concepts of the framework that were identified in the previous chapters.

CHAPTER 5

The In-Motes Testing and Evaluation

5.1 Introduction

The aim of this chapter is to present the results of a series of tests applied to the In-Motes middleware and provide an overall evaluation of the applicability of our system. The testing process which is discussed in detail in the next section was divided into two parts. In the first part, some preliminary tests were executed based on simple applications that were deployed in the wireless sensor network. Also, the merits of the middleware were tested against the Agilla middleware and the TinyOS framework. All those tests were undertaken in the controlled environment of the laboratory. In the second part, the creation of a complex application, namely In-Motes Bins tested the middleware's functionality and applicability under real life conditions in a trial that lasted for four months in a wine store.

5.2 The preliminary tests of the In-Motes middleware

A set of preliminary tests were generated in order to verify the correct execution of the In-Motes middleware main functionalities and correct any possible problems. Moreover, we wanted to test our middleware and its applicability against the Agilla middleware and the stand alone TinyOS framework.

The laboratory experiments were based on a MICA2/DOT Professional Kit (MOTE-KIT 5x4) as shown in Figure 5-1.

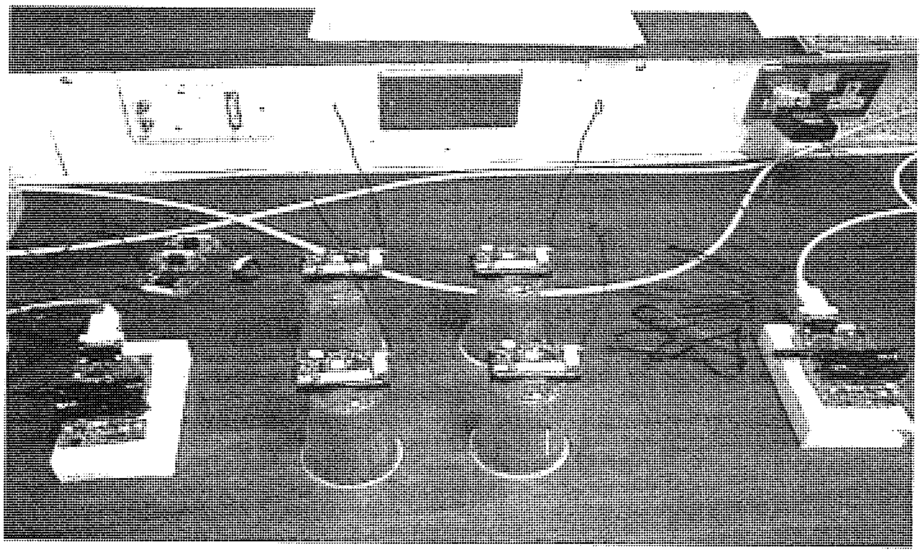


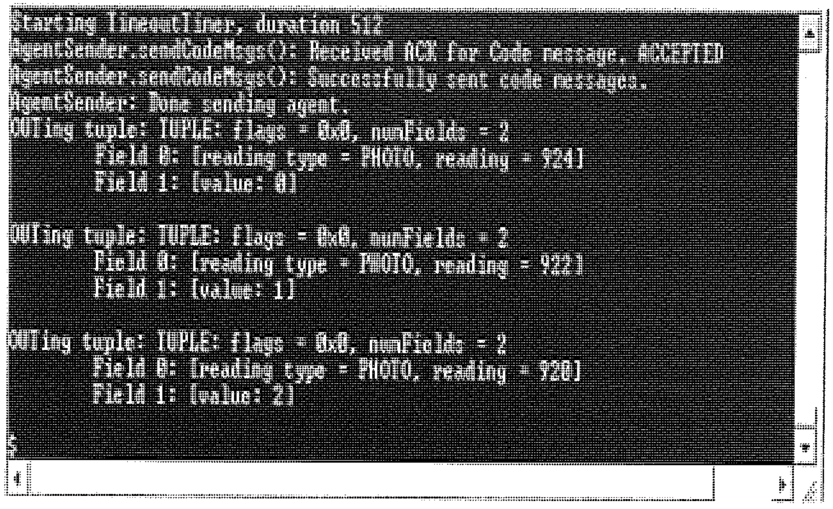
Figure 5-1: Our experimental motes kit with six MICA2 motes and two MIB510 Programming and Serial Interface Boards

Initially we focused our efforts on the correct delivery and functionality of the 4 different mobile codes (In-Motes Agents) in the network that was verified when a node was sending back readings to the user or when it was blinking its LED's in a predefined order set by us. The network consisted of 4 sensor nodes. The facilitator was injected and captured the first available node, in our case the one that was plugged-in in the interface board and then was migrating and storing the ''slave'' instruction to the rest of the available nodes in a random order. Upon arrival and every time a new node was discovered a new tuple was inserted to the newly discovered node's tuplespace having as a reaction the blink of its three LED's for one second. Simple In-Motes job agents were deployed, 45 bytes in size each. The agents were carrying only one instruction for every single sensor which was triggering them to sense the environment and collect a single reading, either light or temperature, that was then send back to the user. The fix In-Motes agent was visiting all the available nodes at once and it was blinking all three LED's of each one of them for one second while in parallel was flashing the internal memory of the network nodes.

Single hopping and multi-hopping migration was also successfully tested by instructing a job agent to blink one of the LED's of the sensors every time they visited a node of the wireless sensor network, allowing us like that to visualize how the agents were migrating to the available nodes. For example, a job agent would visit node 1 and upon collecting the reading it would blink the GREEN led. If the agent is programmed to migrate to node 4, the agent execution is suspended by the In-Motes Agent Manager, it blinked the RED led, and then the code is serialized by the Operation Manager and transferred to node 4, node 1 would blink the YELLOW led and node 4 when the packet arrived would blink the GREEN led.

Since the above tests were mainly based on the predefined blinking of the LED's of the MICA2 nodes a video camera was used for recording the above tests, a number of those videos are available on the In-Motes web site at http://www50.brinkster.com/georgoud/Videos.html

Using the In-Motes de-bugging option we were able at any point to monitor the In-Motes engine that was running in the background by observing the number of acknowledgements and outputs that were produced during the lifetime of an application, Figure 5-2.



```
Starting TimeoutTimer, duration 512
AgentSender.sendCodeMsgs(): Received ACK for Code message. ACCEPTED
AgentSender.sendCodeMsgs(): Successfully sent code messages.
AgentSender: Done sending agent.
OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = PHOTO, reading = 924]
        Field 1: [value: 0]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = PHOTO, reading = 922]
        Field 1: [value: 1]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = PHOTO, reading = 920]
        Field 1: [value: 2]

$
```

Figure 5-2: In-Motes middleware was able to output a set of variables that identified if the software engine was facing any problems

A point of concern appeared when a repeatability test was applied to a single node that was set to report one reading at a time to the base station, in our case temperature, for every job In-Motes agent. Every node was set to uphold all mobile code instructions in a sequence without flashing its memory for every new instruction that was added. This had as result the stall of the mote after the 5th consecutive instruction. Observing the error code that was produced from the middleware, `AgentSender: Fail sending agent`, we concluded, that due to memory limitations every node in the wireless sensor network could not store more than five instructions at a time.

The next step was to create a more complicated scenario that would involve the facilitator framework and protocol and would test the In-Motes engine. For this reason we developed the SenseLight application (Georgoulas & Blow 2006). SenseLight is a multi-hop application that uses mobile code to collect light readings and report back to the end user. The application and small variations of it was deployed in the wireless sensor network by using the In-Motes middleware, the Agilla middleware and the TinyOS framework respectively. Figure 5-3 presents the core differences between the three systems.

Platform	**Communication**	**Network Topology**	**Rules**	**Mobile Code/Agents**
TinyOS	Event Driven	Pre-Programmed	Application Specific	None
Agilla	Tuplespace	Grid Based	Program Specific	Single Agents
In-Motes	Tuplespace with added reactions	Dynamic/ Not pre-programmed	Both application and program specific	Dynamic Mobile code

Figure 5-3: Comparison Table of the three testing platform/middlewares

Both Agilla and In-Motes are using dynamic job mobile code that communicate with the hardware of the sensors using the Linda like tuplespace communication scheme, with the core difference that an Agilla mobile code has multiple roles in the wireless sensor network while In-Motes mobile code has predefined roles and tasks regulated by the facilitator communication scheme. TinyOS on the other hand, is a stand alone event driven platform based on the nesC interface modules that are pre-programmed for every available node of the wireless sensor network prior to any deployment of the application.

The SenseLight application, Figure 5-4, produced two types of In-Motes agents: a Facilitator Agent and a Slave Agent. For the trial we used two facilitator agents and four slave agents. Each slave agent transmitted 1 light reading per message to the facilitator node to which it was associated. Each facilitator agent was instructed to transmit back to the end user 2 light readings per message. The SenseLight application was valid only for data that was produced at an equal rate. Every node in our network was able to handle up to 3 instructions at a time.

```
            // some code is missing…
            // f(applies to facilitator instructions), s( applies to
            // slave instructions)
BEGIN       pushc 26
            jump
            fpushc 28
            fputled // blink yellow led to indicate presence of the
                       agent on the slave node (ACK)
            rjump
            fputled // blink green LED
            sgetvar 0
            scopy
            sinc
            setvar 0
            spushc PHOTO // specify which sensor you are using
            sense // sense the environment
            spushc 2
            spushloc
            fpusht VALUE //collect the value from the slave node
            fpushn abc
            fpushc 2 // template on top of stack
            in       // wait for matching tuple
            fclear       // clear the opstack
            spushc 26
            putled // blink green LED
            frout 1 // remote out tuple containing  first light
                       reading to the laptop
            frout 2 // remote out tuple containing  second light
                       reading to the laptop
            spushc 1
            sleep
            rjump BEGIN
```

Figure 5-4: The SenseLight application that was deployed from In-Motes middleware (Georgoulas & Blow 2006)

Running the Agilla engine in our motes, we deployed the same application with the difference that this time all the agents were reporting back to the end user 1 light reading per message based on the Agilla engine specification and available instruction set.

Removing the middlewares by flashing all the motes we run our SenseLight application on the TinyOS platform. This time, because of the absence of tuple and tuple spaces we made use of the EEPROM that the TinyOS platform offers. The EEPROM serves as a persistent storage device for the mote, and is indispensable for many applications involving data collection, such as sensor data and debugging traces. The EEPROMRead and EEPROMWrite interfaces provide a clean abstraction to this hardware. The EEPROM may be read and written in 16-byte blocks, called lines. Read and write to the EEPROM are split-phase operations: one must first initiate a read or write operation and wait for the corresponding done event before performing another operation.

As can be seen from the below graphs, Figure 5-5 to 5-7, In-Motes produced a higher moving average of 2 light readings to the end-user over a period of 2 minutes. Given a sequence $\{a_i\}_{i=1}^{N}$ of the packets delivered, the n-moving average was a new sequence $\{s_i\}_{i=1}^{N-n+1}$ defined from the a_i by taking the average of subsequences of n terms, which in our case was 120 seconds.

$$s_i = \frac{1}{n}\sum_{j=i}^{i+n-1} a_j. \quad (1)$$

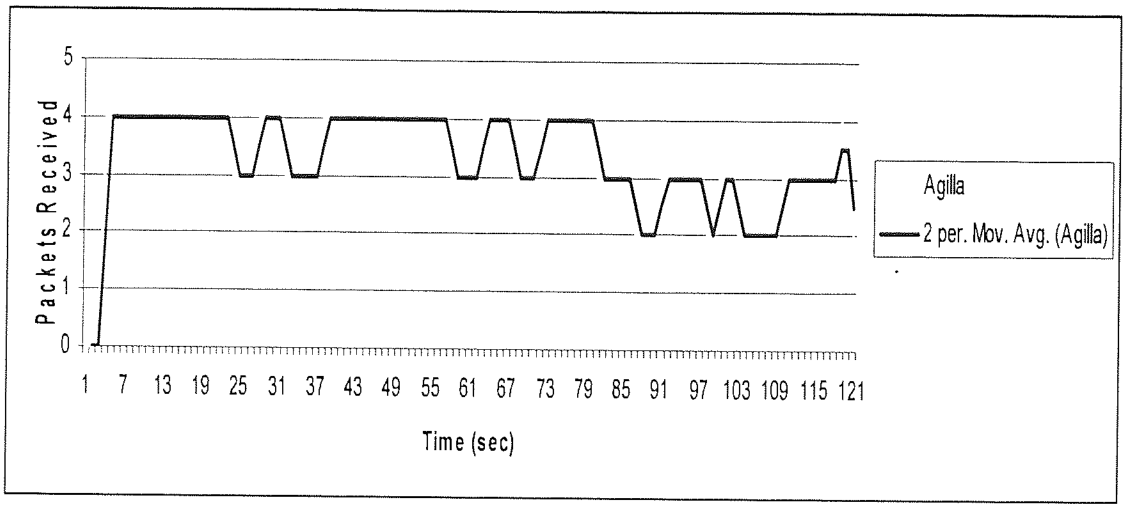


Figure 5-5: Packet Delivery Performance of Agilla over a period of 2 minutes (Georgoulas & Blow 2006)

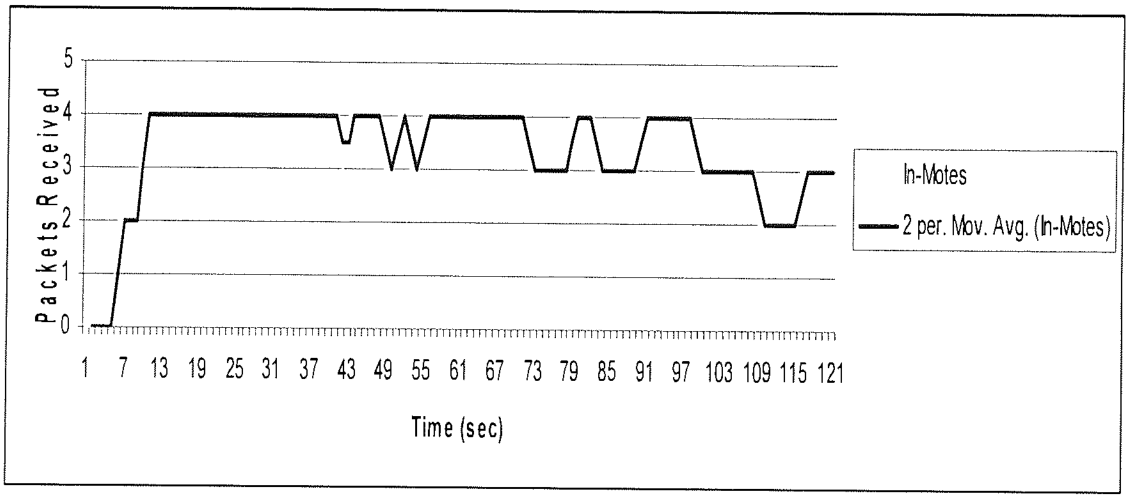


Figure 5-6: Packet Delivery Performance of In-Motes over a period of 2 minutes (Georgoulas & Blow 2006)

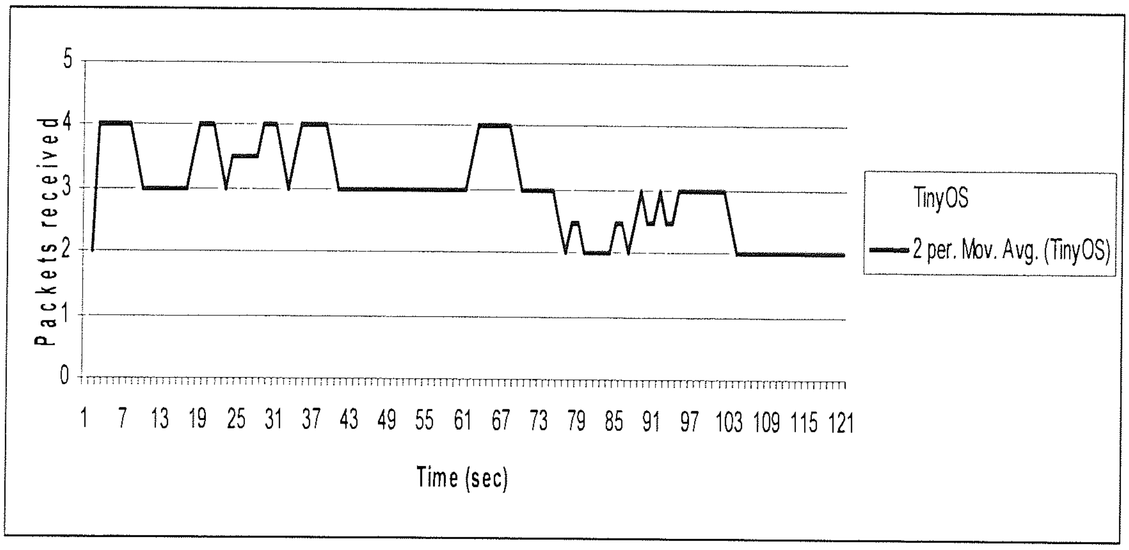


Figure 5-7: Packet Delivery Performance of TinyOS over a period of 2 minutes (Georgoulas & Blow 2006)

We conclude that the use of a middleware has significantly improved the performance in packet delivery. Both In-Motes and Agilla deliver better results than of those produced by the TinyOS platform. In-Motes, based on the facilitator scheme, was able to create a better performance rate than Agilla. The throughput of the In-Motes middleware in the above graphs seems to vary and at some point to increase. This behaviour was based on the fact that initially the facilitator nodes were processing acknowledgements from the slave nodes, which they captured, that were ready to receive requests. Each acknowledgment was a 4 bytes packet. Thus, there was a small delay in obtaining the results initially but as soon as this was satisfied the facilitators were reporting light readings back to the terminal, although this process produced some delays in obtaining the data, it guaranteed that all nodes of the wireless sensor network were active and communicating with the middleware.

All three systems as can be seen from the above graphs showed similar behaviour in that after a time, on average more than 3 minutes of continuous transmission, the rate of successful light readings delivery was reduced, mainly based on the unreliable low-bandwidth radio that leads to transmission failures and packet losses. The longest period that the application was set up running was 10 minutes during which the packet delivery performance was varying for all three systems in a similar way with the displayed graphs above. No catastrophic scenarios in terms of total network failure were observed. On average, for all three systems, after the first 100 seconds the decrease in performance was sustained and remained roughly constant on every trial even for the longest periods of continuous transmission (10 minutes). This was a first indication that our motes, independent of the platform that were assigned to, were subject to failures of transmission leading to packet losses, especially when they were constantly active without any timeouts or sleep stages. The success rate of packet delivered after the end of this trial was 65%, 57% and 55% for In-Motes, Aggila and TinyOS respectively. Working with motes such as the ones we used you can observe and identify failures only at the receiving and transmitting end. One of our first lessons learned was that it was almost impossible to identify errors in between those two ends during a transmission.

Another hurdle that was identified through the above experimental process was the build in specification of the motes. Thus, identifying a fault in a mote and recognizing the reason behind it, were two different things with reasons extending from limited battery power to packet losses and internal race conditions. The abstract character that the In-Motes middleware gives to the wireless sensor network and the ability to observe at any point the actual behavior of the middleware engine proved really helpful in order to identify and explain unexpected behaviors from the sensors of our system.

The last part of the laboratory tests include a repeatability test between the three approaches we described above. The test evaluated the total number of packets received by the base station in a period of two minutes for 5 consecutive sessions during which we point out that the sensing is not scheduled but the sampling/wait is due to periodic request.

Figure 5-8 demonstrates the results of this test. Once again In-Motes middleware had a better performance although due to errors mainly concerned with packet losses and memory limitations of the individual sensors this performance wasn't steady. More precisely, In-Motes had variability in the results between each repetition of ± 6 packets slightly better than that of Agilla that had ± 7. TinyOS maximum variability was ± 10 packets. Also, In-Motes in all five repetitions of the testing session had the higher number of packets delivered although it used 2 motes less, occupied by the corresponding facilitators, than Agilla and TinyOS thus demonstrating better use of the available motes of the wireless sensor network.

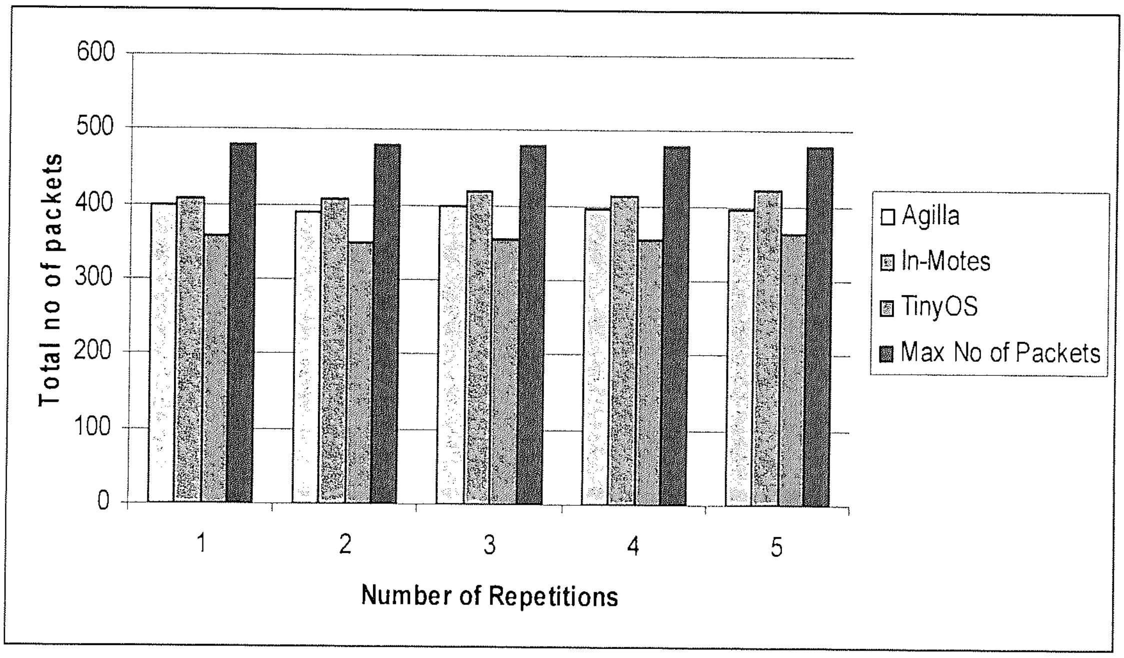


Figure 5-8: Repeatability chart performance of In-Motes, Agilla and TinyOS for 5 consecutive sessions (Georgoulas & Blow 2006)

5.3 The In-Motes Bins application

Having identified certain behaviors for both our middleware and the wireless sensor network, we wanted to test the merits of our software in a real time environment where the wireless sensor network would be applied in a busy everyday location. The instability and the frequent errors of the network that were reported in the laboratory initial trials was of a major concern and we were aiming to identify if this behavior was generic and due to the character of the sensor nodes or if noise and interference from other electronic equipment in the laboratory made the wireless sensor network susceptible to errors.

5.3.1 The wine merchant store

The wine merchant store is located in one of the busiest roads in Birmingham, Hagley Rd and it is one of the largest ones in the West Midlands. Permissions were acquired from Miss Jude Fenner, area manager, and the shop manager Mr. Cameron Foreman to allow us to set up our wireless sensor network in the premises for a period of four months. We had access to the whole store including staff areas during the opening hours, as well as at the weekends. Based on that, we were able to set up the network and run the application during a specific time period. Thus, we were not able to have the network active during the night.

The store can be divided in two main areas, the public one which is the main floor and consists of shelves and various wine displays along the floor and a store room containing a huge number of wine boxes.

When we visited the store we were informed about a problem that they were facing affecting them both financially and ethically, the problem of what they called in generic terms, "corked" wines. An interview with the assistant manager, Mr. James Gormley, gave us a better idea about the product, the problem they were facing and the environment that we would apply our application (Appendix B). According to Larsson and Spittler (1990) wine is a product which is very sensitive to light and temperature variations. Wine must be stored in temperatures between 5°C up to 18°C. The degree and the speed of the temperature changes are critical as large temperature changes or fluctuations will oxidize the wine. Also light/direct light is another factor for ageing the wine faster. Thus, our trial involved the investigation of which areas of the wine store were inappropriate for storing/displaying wine. The outcome of these tests would be reported to the store manager. Figure 5-9 identifies the problem with the "corked" wines which we were aiming to explain and provide a solution through our trial.

Figure 5-9: "Corked" wines were a frequent phenomenon at the store prior to our visit

Thus, we developed the In-Motes Bins application, a mobile code real time application developed for monitoring environmental conditions (Georgoulas & Blow 2007). In terms of hardware we have used in total a set of 10 mica2 sensors with the accompanied MTS310CA sensor boards. One base station, a laptop connected with an MIB520CA interface board fixed in a secure area of the store served as the aggregation point. All the hardware was provided by the Crossbow Technology Ltd. All the sensor motes were working under the TinyOS operating system and the application was deployed through our In-Motes middleware, Figure 5-10.

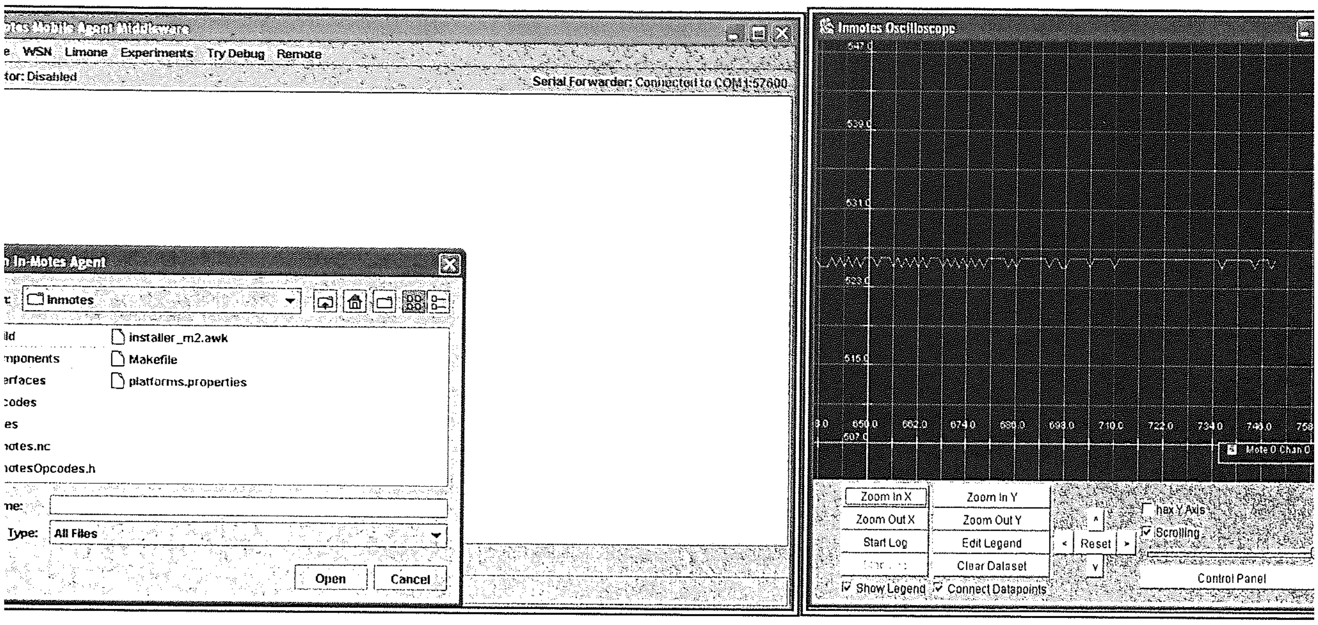


Figure 5-10: The In-Motes V1.0 GUI and the Oscilloscope Interface that were used for the In-Motes Bins Trial

5.3.2 The In-Motes Bin design

Our trial was divided into two areas as we are going to demonstrate in the next sections. Between the months June 06 and July 06 the In-Motes Bins application was applied exclusively to the warehouse of the store while between August 06 and September 06 the wireless sensor network and the application were deployed in the main floor of the store. As we mentioned earlier we had access to the store only during the opening hours and not during the night.

To deploy the In-Motes Bins application in the warehouse we used in total 3 mica2 sensors and one interface board connected with the laptop which served as the aggregation point. One facilitator agent was responsible for capturing the available nodes of the wireless sensor network and all the queries were forwarded using static job agents (Georgoulas & Blow 2007).

The facilitator captured the nodes and was ready to receive the first request in 15sec. We were sending static temperature job In-Motes agents, which did not provide any local decision making to the wireless sensor network every 30min and the facilitator was reporting readings from its slaves back to the base station for a period of 1min.

The job In-Motes agents were transferred to the wireless sensor network by cloning. Thus, each node was assigned with a residing static job agent for the duration of the measurement period, consuming 68 bytes of the available memory. Before we re-deployed the application, we flushed the slave nodes by removing the temperature job tuples in order to have enough memory space.

In two months we collected 64800 readings from the slave nodes in the warehouse giving a very clear picture of the temperature variations that were taking place. Figure 5-10, demonstrates the In-Motes Bins application code that was deployed to the wireless sensor network during the warehouse trial.

```
           // some code is missing...
BEGIN      fpushloc 3
           fceq
           frjumpc CLONE 3 // make 3 clones of the agent
           CLONE 3
           fpushloc 3
           fclone
           halt
           CLONE 3
           fpushloc 3
           fclone
           CLONE 3
           fpushloc 3
           fclone
           halt // stop cloning
           sputled // blink the green LED
           sgetvar 0 // increment the counter stored in heap[0]
           scopy
           sinc
           setvar 0
           spushc TEMP // specify which sensor you are using
           sense // sense the environment
           spushc 2
           spushloc
           srout        // remote out tuple containing temperature
                      //reading to the facilitator
           fpushc 1
           frjump BEGIN
```

Figure 5-11: The In-Motes Bins application that was deployed from In-Motes middleware during the warehouse trial

In order to deploy the In-Motes Bins application in the main floor we used all our available mica2 sensors. For reasons that we are going to present next and explain further in the analysis section of this chapter we have changed the In-Motes Bins specification, with the application this time having core design differences with the one that was deployed in the warehouse.

Firstly, we used dynamic job agents for forwarding all the queries in order to provide a more energy efficient way for the motes to present measurements by eliminating re-deployments (Georgoulas & Blow 2007). Every job request that was forwarded to the wireless sensor network provided a specific critical parameter used to locally determine if a measurement should be reported. Thus, the total collected readings would be less than before and the data more meaningful making our analysis in the end far easier.

Secondly, the dynamic In-Motes job agents were transmitted in the wireless sensor network by migration rather than cloning. For the warehouse trial and since the application involved only three sensors, cloning was sufficient enough but still quite energy consuming. Considering the scale of the network in the main floor and the fact that we were trying to conserve as much energy as possible, agent migration in terms of forwarding user queries and collecting results was the right decision.

The dynamic In-Motes job agent was sent once inside the network and was capable of local decision making. The dynamic In-Motes job agent works by inserting a job tuple upon arrival. The job tuple this time apart from the string containing the job description will also contain the string 'crt' which matches the specific critical parameter for the job. As before if the behavioral rules of the facilitator are satisfied the dynamic job agent will migrate to all its slaves. The slaves wake up every 30 minutes and sense one reading. If the reading is above the critical parameter then the slave node will transmit readings for 1 minute and then it will sleep until the next wake up time (Georgoulas & Blow 2007).

Figure 5-11, demonstrates part of the In-Motes Bins application code that was deployed to the wireless sensor network during the warehouse trial.

```
// Variable Declaration code omitted
spushm TEMP
spushm CRT
sense          // sense temperature or light
IF          TEMP > CRT THEN
spushm 2
spushloc
rout           // remote out tuple containing temperature reading
            //to facilitator
spushm 1
sleep // for 1 minute
ELSE
sleep // for 1 minute
// some code is missing …
rstart BEGIN
```

Figure 5-12: Part of the In-Motes Bins application that was deployed from In-Motes middleware during the main floor trial (Georgoulas & Blow 2007)

We initialized our wireless sensor network by injecting once two In-Motes facilitator agents with four nodes acting as slaves to each of them. The facilitators captured the nodes and were ready to receive the first job requests in an additional 35s based on the fact that the capturing sequential procedure was repeated twice with an equal amount of slave nodes, making the total time of 50s. We send, once, two dynamic job agents to the wireless sensor network.

The above procedures, as well as the wake up calls and the transmission of data readings were executed by the application without us interfering with the process unless a failure was noticed. Since this time our application was autonomous, no flushing of the slave nodes took place unless a network failure occurred. Using the dynamic In-Motes job agent and the facilitator agent, the wireless sensor network could report temperature and/or light readings, if the two facilitator nodes were instructed to work independently by allocating to each of them a different job agent.

5.4 The In-Motes Bins field tests

In the next sections we are going to provide an analytical report of the findings of the field trial that lasted for four months in a wine chain store (Oddbins) between June and September of 2006.

5.4.1 In-Motes Bins: The Warehouse

The warehouse was the perfect location to start the trial and deploy our In-Motes Bins application based on facts such as its business in terms of people interactions and size as it was smaller than the main floor. Thus, we were able to set up a small wireless sensor network, using initially half of our sensors and deploy our application measuring only temperature variations since the warehouse had at all times constant artificial lighting due to absence of any windows.

The warehouse was divided in three sections with a single sensor placed on the highest point of each section. The aggregation point was placed in the office that existed in the warehouse. Figure 5.13

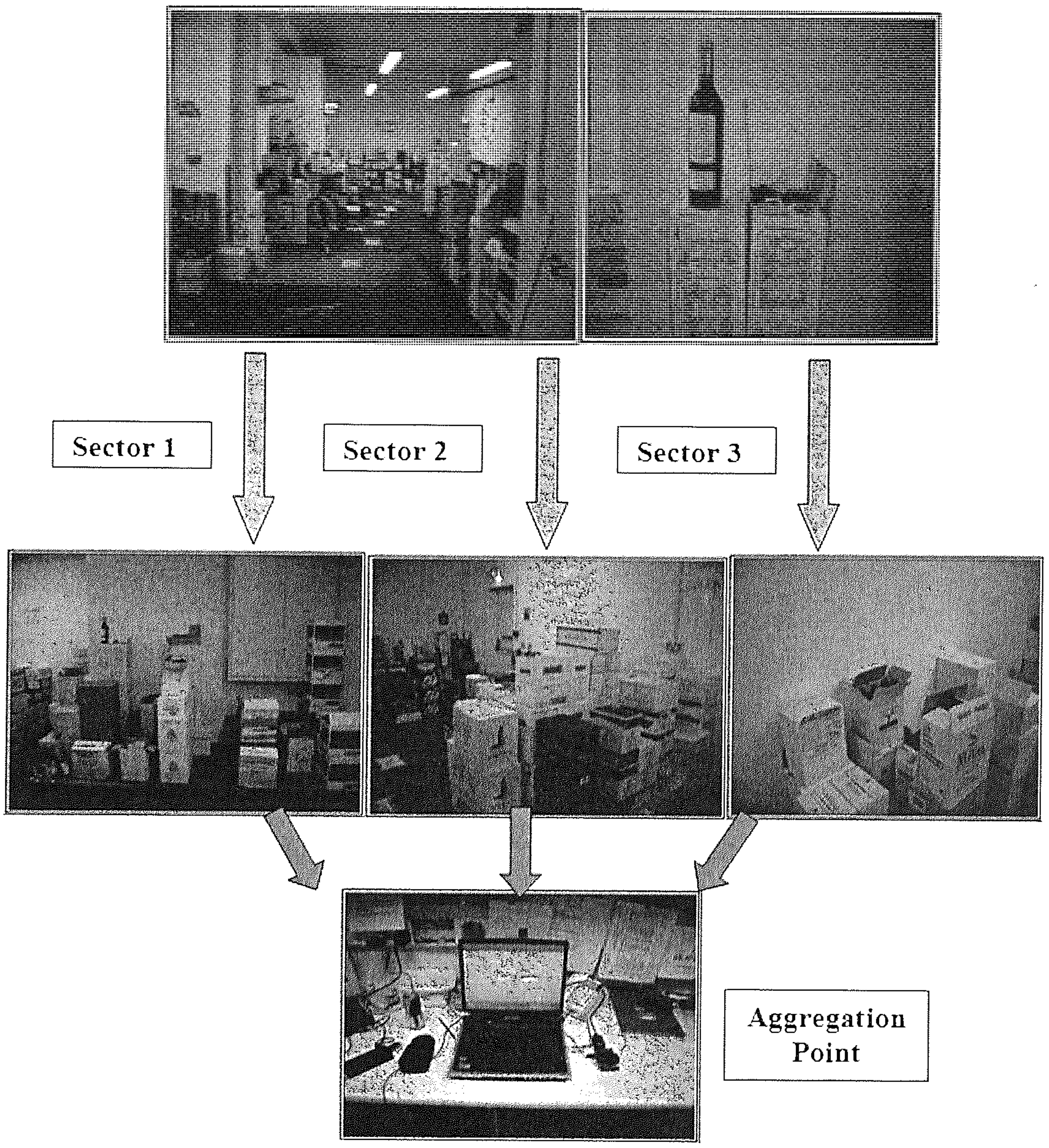
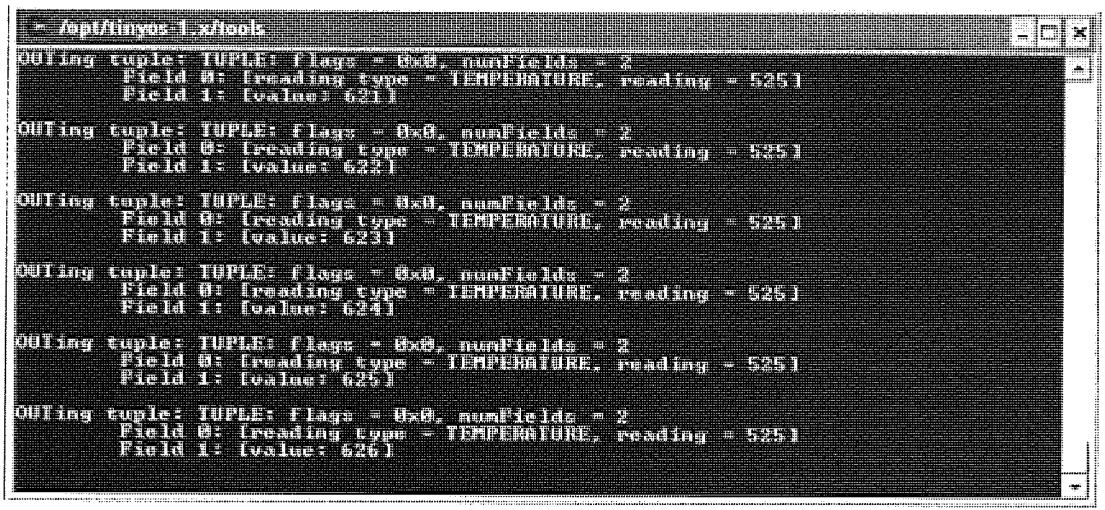


Figure 5-13: The warehouse was decomposed in three sections with one sensor each reporting back to the interface board at the aggregation point

The mote's output, Figure 5-14, was converted to Kelvin degrees using the following approximation over 0-50 °C:

$$1/T(K) = a + b X \ln(Rthr) + c X [\ln(Rthr)]3 \quad (2)$$

Where:
Rthr = R1(ADC_FS-ADC)/ADC
a = 0.00130705
b = 0.000214381
c = 0.000000093
R1 = 10 kΩ
ADC_FS = 1023
ADC = output value from Mote's ADC measurement



```
~ /opt/tinyos-1.x/tools
OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = TEMPERATURE, reading = 525]
        Field 1: [value: 621]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = TEMPERATURE, reading = 525]
        Field 1: [value: 622]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = TEMPERATURE, reading = 525]
        Field 1: [value: 623]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = TEMPERATURE, reading = 525]
        Field 1: [value: 624]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = TEMPERATURE, reading = 525]
        Field 1: [value: 625]

OUTing tuple: TUPLE: flags = 0x0, numFields = 2
        Field 0: [reading type = TEMPERATURE, reading = 525]
        Field 1: [value: 626]
```

Figure 5-14: Display of temperature readings from the sensor in section 1 of the warehouse (Georgoulas & Blow 2006)

Section 1

Section 1 was quite close to the office and to the warehouse door. Thus, its sensor was the closest one to the aggregation point, 10.2 meters, and also the one that would have encounter the biggest volumes of outside interference. By this term we mean the members of the staff that would move quite frequently around it. Due its location we weren't expecting really high temperatures since the air seemed that was circulating enough.

On average we were collecting 250 temperature readings per day from the sensor that was placed in Section 1. Figure 5-15 and 5-16 display the maximum temperatures that Section 1 reported during two days in the third week of June. The straight line in the graphs identifies the borderline of 18 °C, anything above that temperature and the products would have been affected.

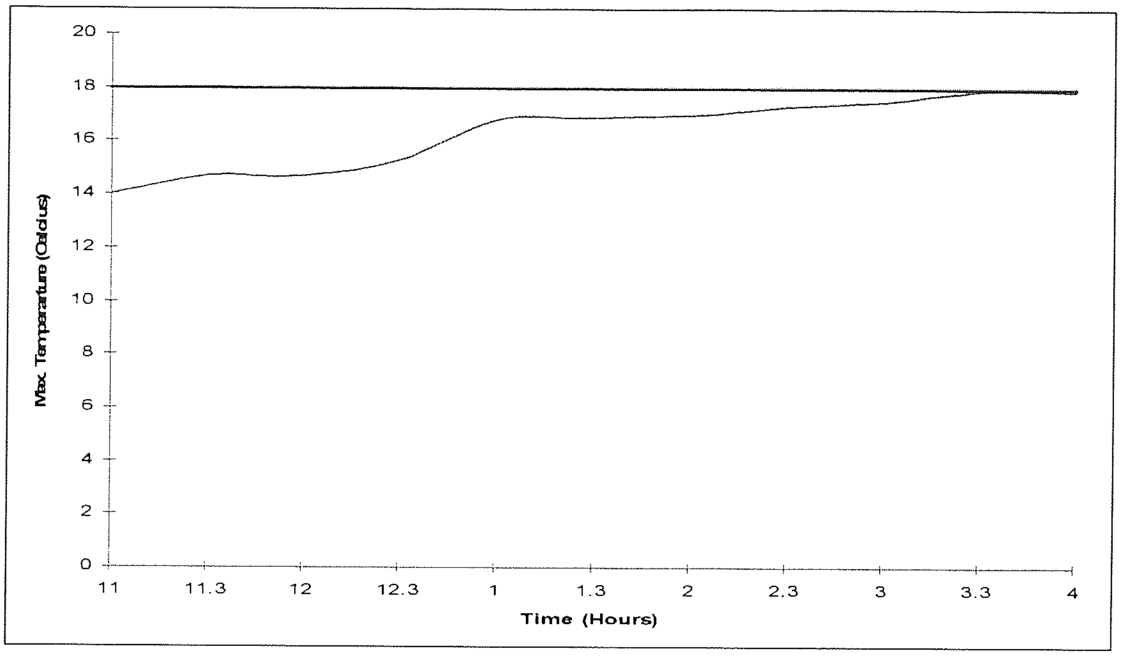


Figure 5-15: Section 1 maximum temperature readings during 20/06/06

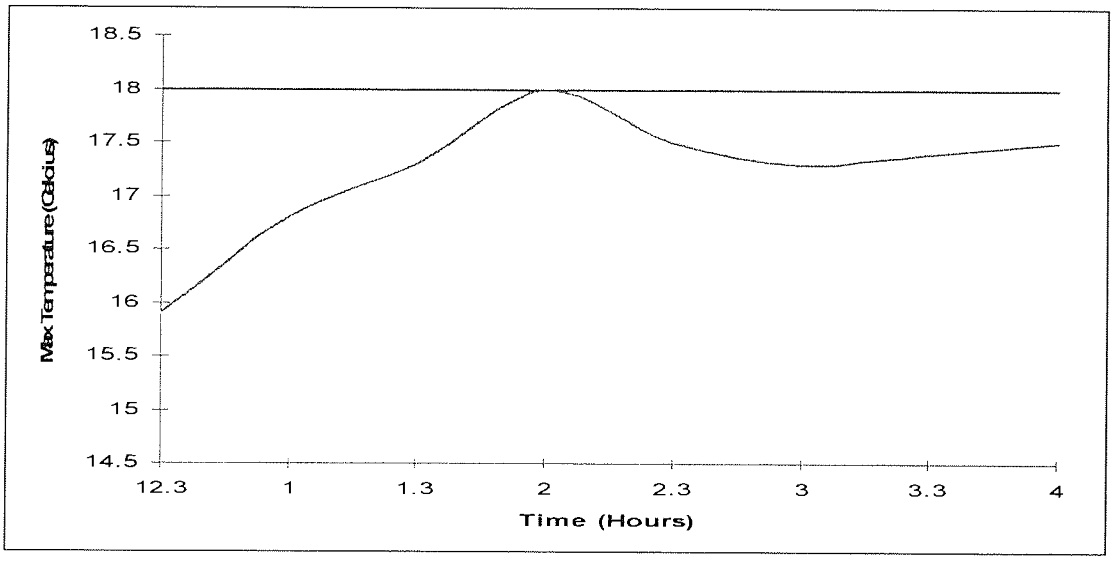


Figure 5-16: Section 1 maximum temperature readings during 24/06/06

From the above graphs we identify that Section 1 was reporting quite high temperature variations but without overcoming the borderline of the 18 °C. These graphs are representative of the temperature readings we collected during that month. Thus, the wines that were stored there suffered from those temperature fluctuations in a long term and only if they were located there for more than one month the quality of the product would be reduced.

A similar behavior was reported during July with the main difference that the fluctuations were higher than those of June and the maximum readings were closer to the borderline of the 18 °C as shown in Figure 5-17.

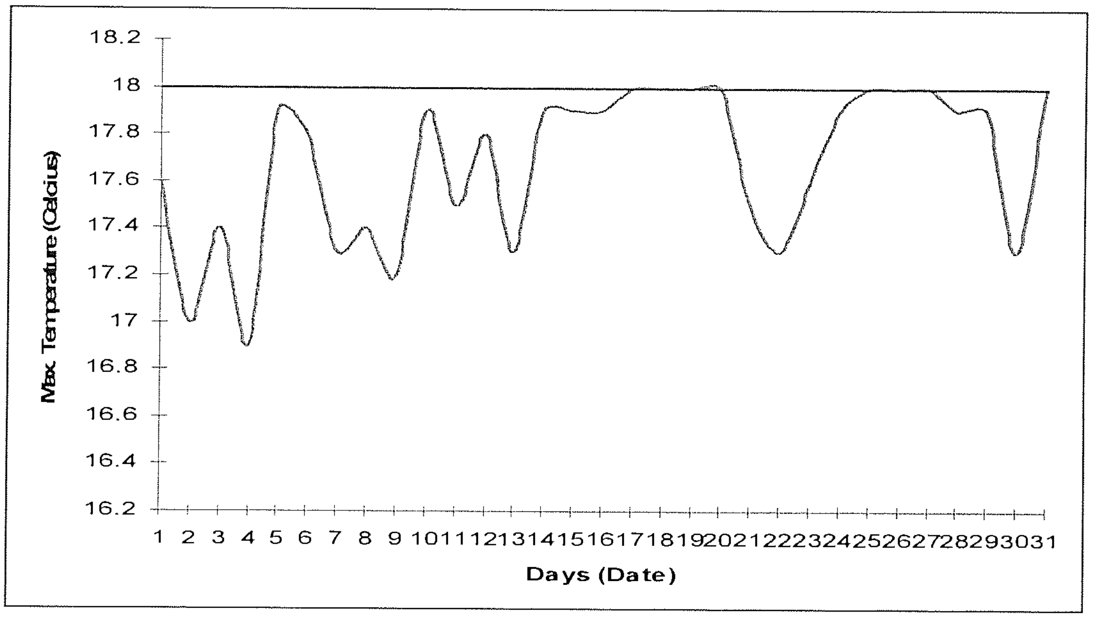


Figure 5-17: Section 1 maximum temperature readings during July

<u>Section 2</u>

Section 2 was located in the middle of the warehouse and on the right, 13.72 meters away from the aggregation point. It was relatively far away from the warehouse door so although we were expecting less interference from the employees, the air did not circulate equally well as in Section 1 making us to predict that we would monitor higher temperatures.

Figure 5-18 demonstrates the maximum temperature readings that the sensor was reporting in Section 2 during July. We were right with our predictions, as this time the temperatures were raising even above 18 °C. From this analysis it was obvious that products that were stored in that section if they were not placed soon enough in the main floor they would be affected relatively easy.

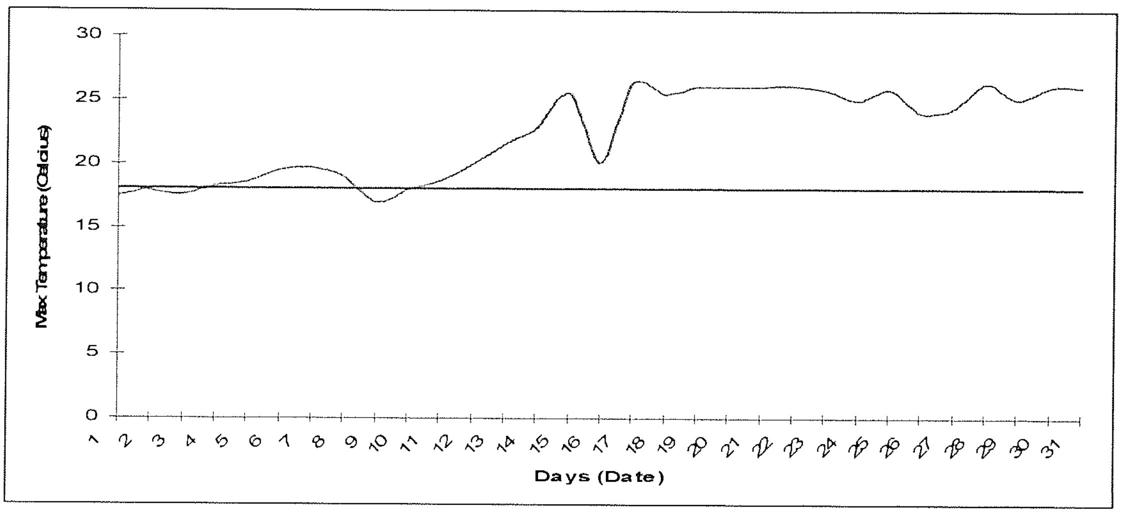


Figure 5-18: Section 2 maximum temperature readings during July

Section 3

Section 3 was located towards the end of the warehouse next to the fridge and underneath a powerful fridge motor. The sensor was placed 16 meters away from the aggregation point and it was not in the line of sight of the interface board. Also it was the section of the warehouse where there was a lot of human activity and movement of boxes.

Figure 5-19 demonstrates the maximum temperatures that the sensor of Section 3 reported during July. We were impressed as temperatures reached even to levels above 30°C making the area totally inappropriate for storing any kind of wine.

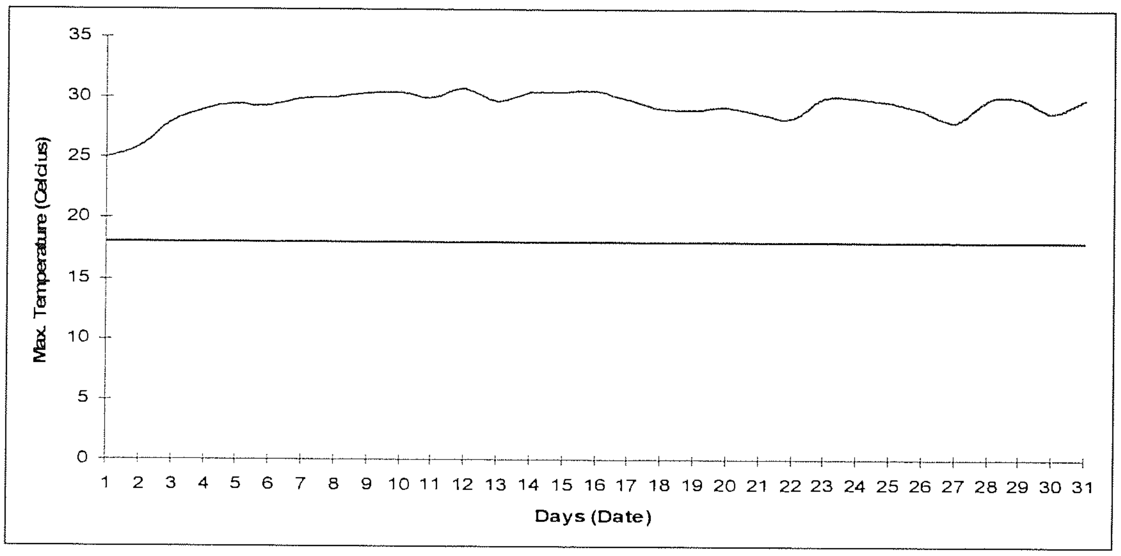


Figure 5-19: Section 3 maximum temperature readings during July

In terms of identifying the bad sectors for storing wine in the warehouse our analysis of the results provided us with a really good view of the problem that the store was facing. Especially Section 3, where the fridge motor engine that existed on top of the boxes was generating extremely high temperatures. Our trial analysis was forwarded to the manager and as can be seen for Figure 5-20 changes took places in the warehouse after the end of our trial.

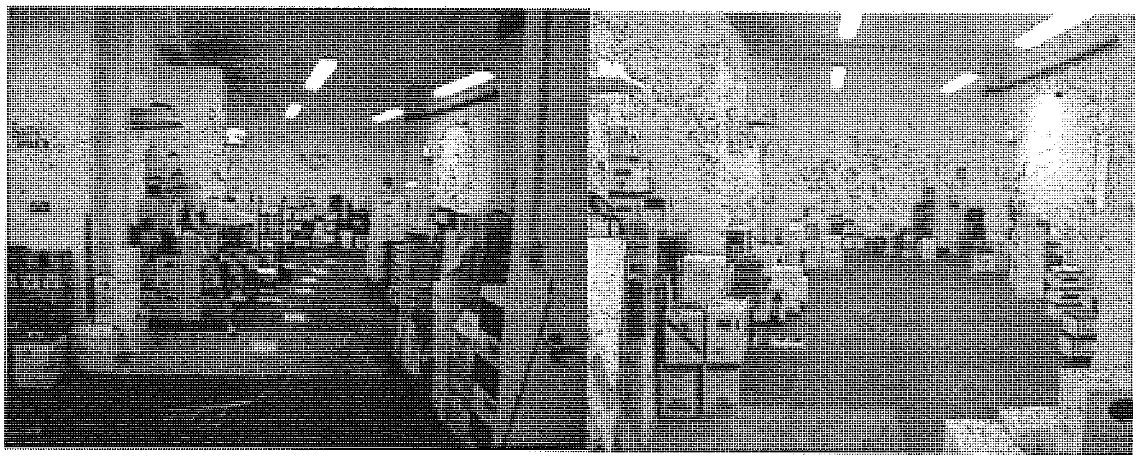


Figure 5-20: The warehouse before (left) and after (right) the finish of our trial (Georgoulas & Blow 2007)

5.4.2 In-Motes Bins: The main floor

The In-Motes Bins application was deployed in the main floor of the shop during August and September and as we mentioned above the application had significant changes from the one that was applied in the warehouse.

In total the main floor was divided into 8 sections, one facilitator node for every four nodes, as can be seen from Figure 5-21.

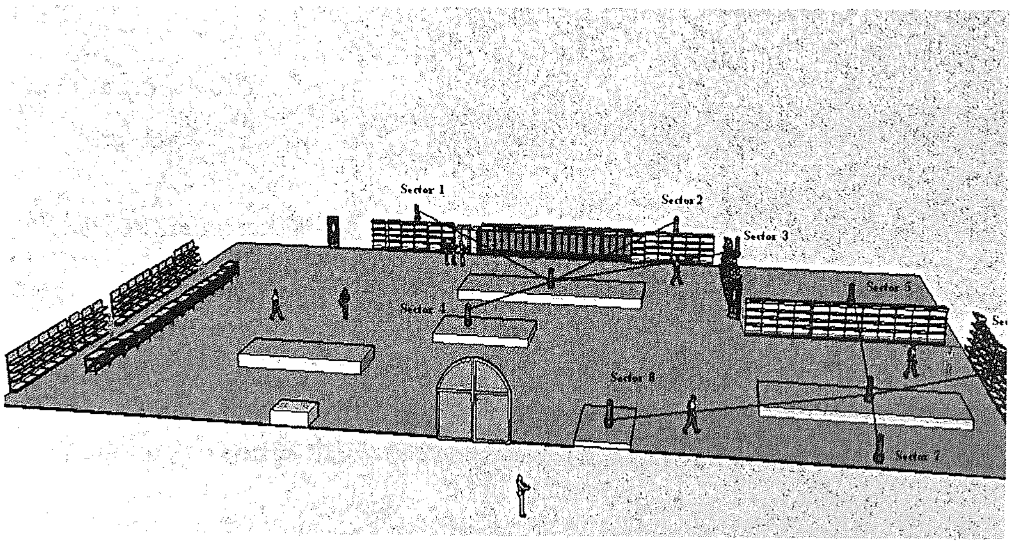


Figure 5-21: A 3D representation of the 8 sections of the main floor

A mica2 sensor was also placed outside the store in order to observe the outside temperature changes creating a better comparison model, Figure 5-22. This node was communicating directly with the interface board at the aggregation point and it was accessed at random intervals. The wireless sensor network was reporting either temperature or light readings for most of the days. At random time intervals in order to examine the flexibility of our middleware we were assigning one facilitator and its "federation" of nodes to report light readings while the other facilitator was continuing reporting temperature readings.

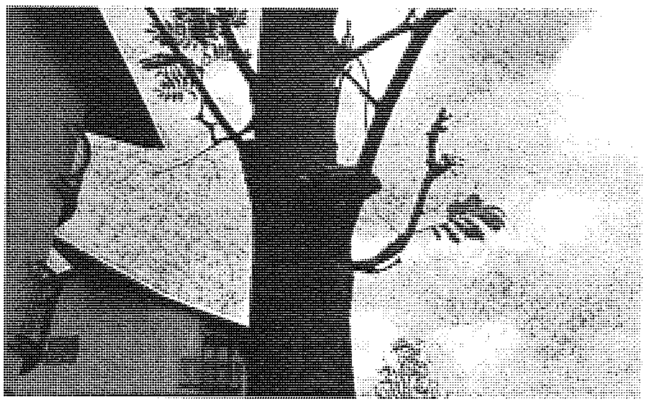
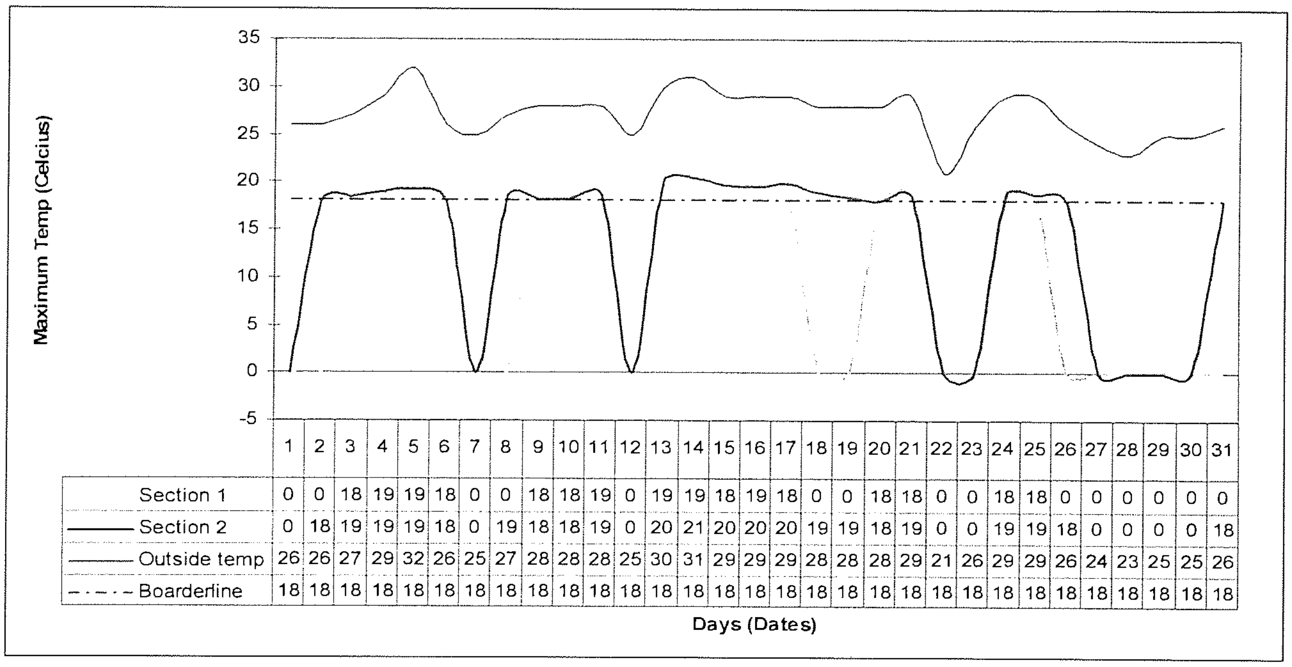


Figure 5-22: A mica2 sensor was placed on a top of tree providing us with outside temperature readings

Temperature Analysis

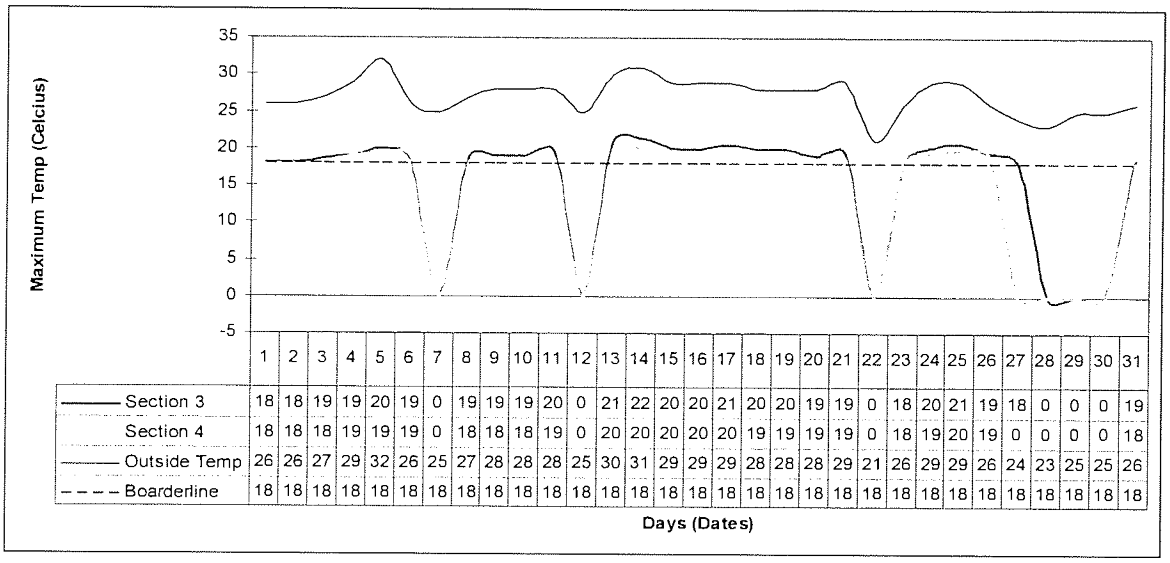
The facilitator node 1 was set to collect temperature readings from sections 1-4 and the distance from the aggregation point was 14.5 m. The distance between the facilitator node and its slave's nodes was less than 2 m. Figure 5-23 and 5-24 shows the maximum temperatures that were recorded during August in all four sections.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 1	0	0	18	19	19	18	0	0	18	18	19	0	19	19	18	19	18	0	0	18	18	0	0	18	18	0	0	0	0	0	0
Section 2	0	18	19	19	19	18	0	19	18	18	19	0	20	21	20	20	20	19	19	18	19	0	0	19	19	18	0	0	0	0	18
Outside temp	26	26	27	29	32	26	25	27	28	28	28	25	30	31	29	29	29	28	28	28	29	21	26	29	29	26	24	23	25	25	26
Boarderline	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18

Figure 5-23: Section 1 and 2 maximum temperature readings during August

Both Sections as can be seen were reporting temperatures above 18 °C at some point during August with Section 1 alerting the system fewer times than Section 2. The temperature fluctuation pattern for both sections that was produced was quite similar with the outside temperature changes.

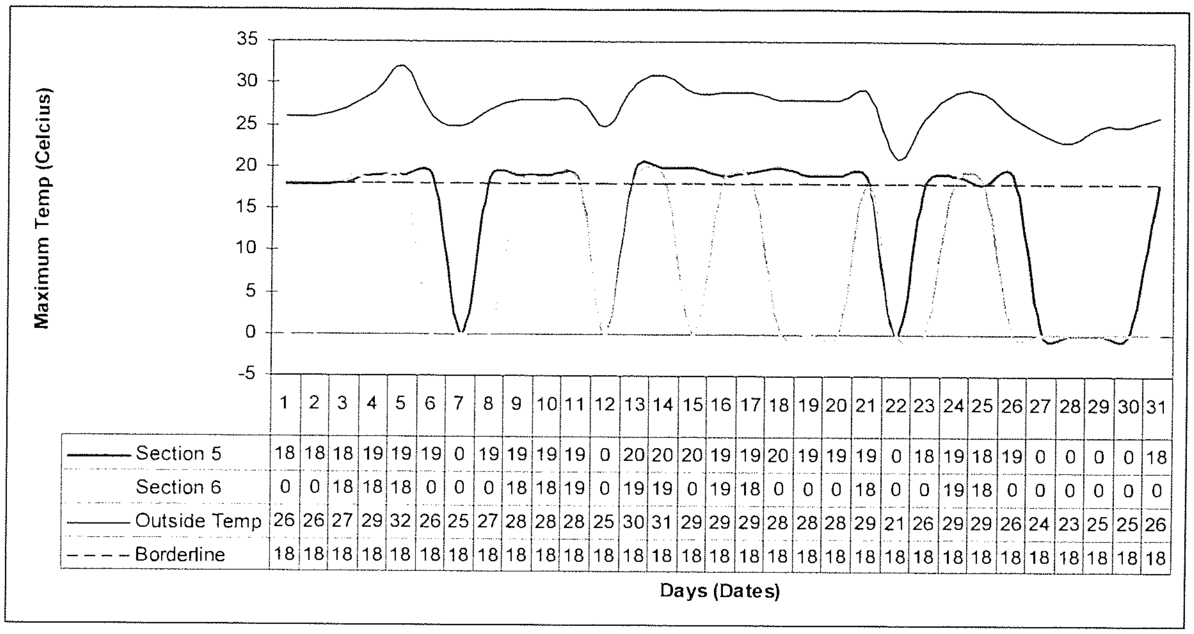


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 3	18	18	19	19	20	19	0	19	19	19	20	0	21	22	20	20	21	20	20	19	19	0	18	20	21	19	18	0	0	0	19
Section 4	18	18	18	19	19	19	0	18	18	18	19	0	20	20	20	20	20	19	19	19	19	0	18	19	20	19	0	0	0	0	18
Outside Temp	26	26	27	29	32	26	25	27	28	28	28	25	30	31	29	29	29	28	28	28	29	21	26	29	29	26	24	23	25	25	26
Boarderline	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18

Figure 5-24: Section 3 and 4 maximum temperature readings during August

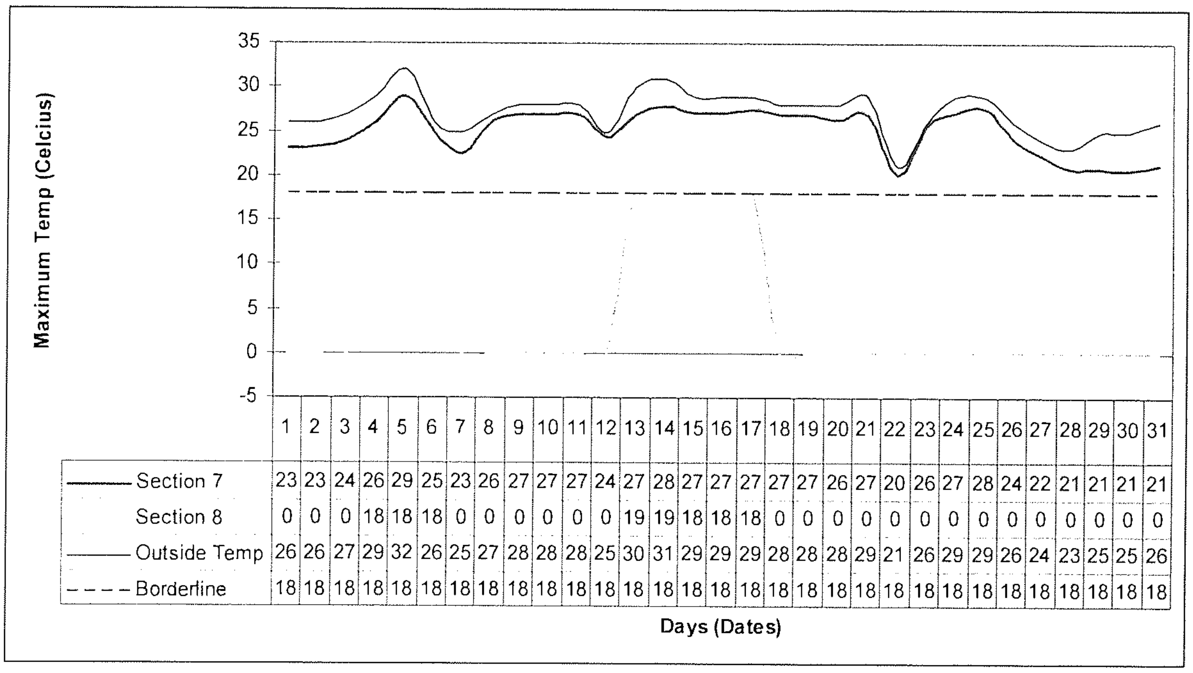
Section 3 was by far reporting the highest temperatures than any other section of the facilitator node 1 and its sensor alerted the system more times than any other sensor of the group. As before both Sections 3 and 4 were following a similar fluctuation pattern as this of the outside temperature.

The facilitator node 2 was set to collect temperature readings from sections 5-8 and the distance from the aggregation point was 55.5 m. The distance between the facilitator node and its slave's nodes was as before less than 2 m. Figure 5-25 and 5-26 shows the maximum temperatures that were recorded during August in all four sections.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 5	18	18	18	19	19	19	0	19	19	19	19	0	20	20	20	19	19	20	19	19	19	0	18	19	18	19	0	0	0	0	18
Section 6	0	0	18	18	18	0	0	0	18	18	19	0	19	19	0	19	18	0	0	0	18	0	0	19	18	0	0	0	0	0	0
Outside Temp	26	26	27	29	32	26	25	27	28	28	28	25	30	31	29	29	29	28	28	28	29	21	26	29	29	26	24	23	25	25	26
Borderline	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18

Figure 5-25: Section 5 and 6 maximum temperature readings during August



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 7	23	23	24	26	29	25	23	26	27	27	27	24	27	28	27	27	27	27	27	26	27	20	26	27	28	24	22	21	21	21	21
Section 8	0	0	0	18	18	18	0	0	0	0	0	0	19	19	18	18	18	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Outside Temp	26	26	27	29	32	26	25	27	28	28	28	25	30	31	29	29	29	28	28	28	29	21	26	29	29	26	24	23	25	25	26
Borderline	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18

Figure 5-26: Section 7 and 8 maximum temperature readings during August

Section 5 and 6 were following the pattern of the other sections with temperatures rising above 18 °C especially during the hot days of the month. Section 7 was the worst section and the most active of the whole main floor with temperatures above 20 °C and fluctuations almost identical to the outside ones. This can be explained as this section was the closest one to the front window of the shop. The lowest temperatures and alerts were observed in Section 8. This section although quite close to the shop windows it had the advantage that it was next to the main door and only 2 m away from the air-conditioning unit. During our trial in the main floor the air-condition unit was set constantly at 20 °C.

An additional problem of the store in terms of its temperature fluctuations was this of the wooden selves and displays, Figure 5-27.

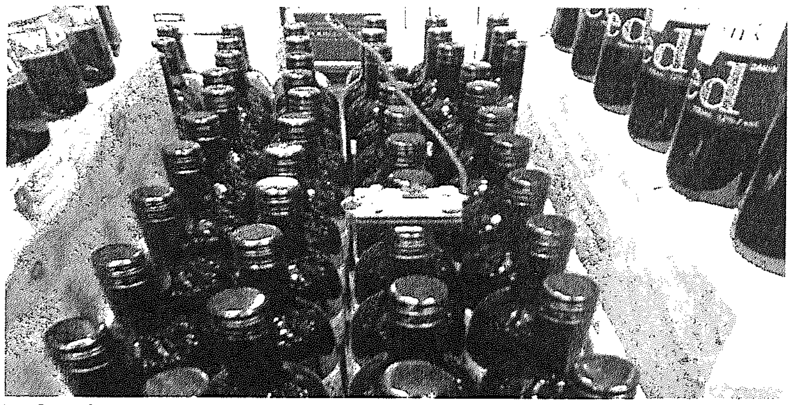


Figure 5-27: Section 7 that reported the highest temperature readings. Wines were displayed on wooden boxes that were cooling down very slowly (Georgoulas & Blow 2007)

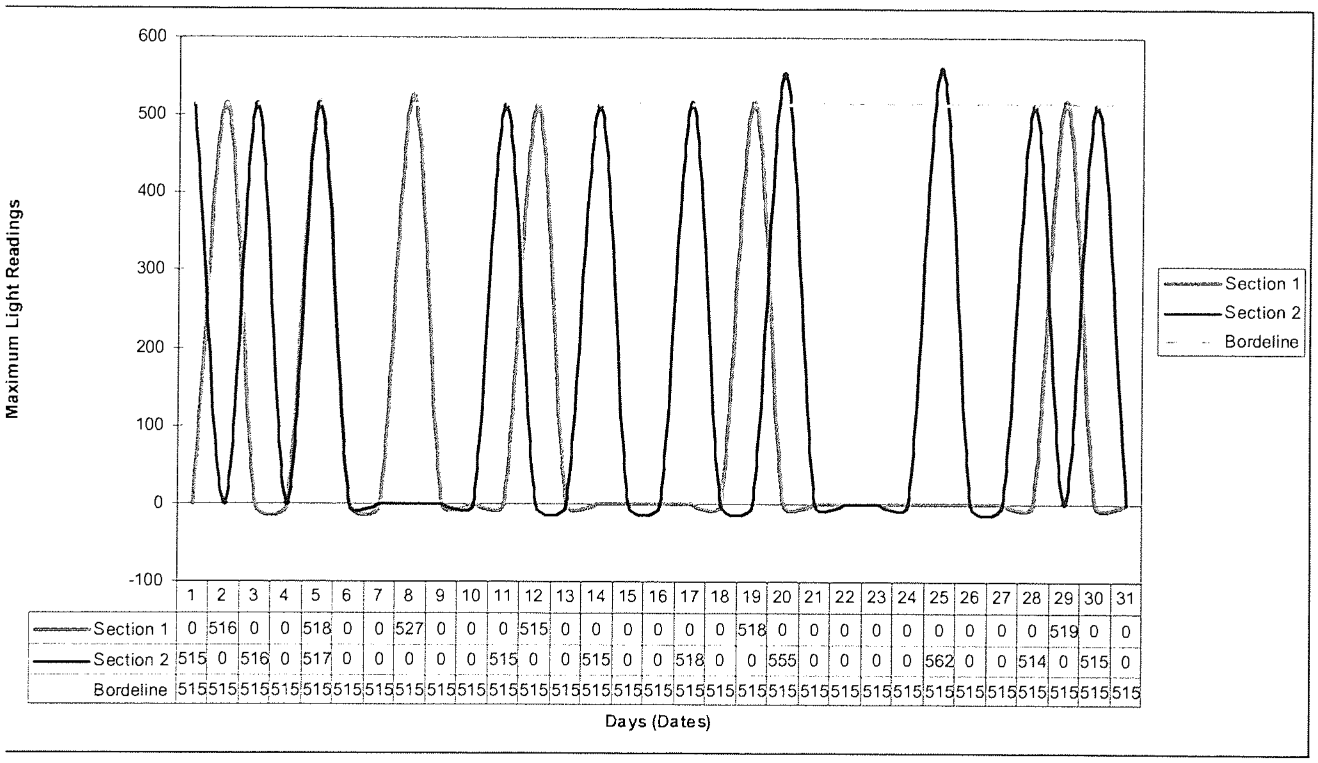
During the summer months almost every single wine of the main floor was exposed to temperatures that were not appropriate and it was almost certain that wines that would have stayed for too long especially on sections such as 4 and 8 would have been corked and aged faster.

Light Analysis

The collection of the light readings was executed in an identical way as the temperature one. The light sensor is a simple CdSe photocell. The maximum sensitivity of the photocell is at the light wavelength of 690 nm. When there is light, the nominal circuit output is near VCC or full scale, and when it is dark the nominal output is near GND or zero (Crossbow manual 2004).

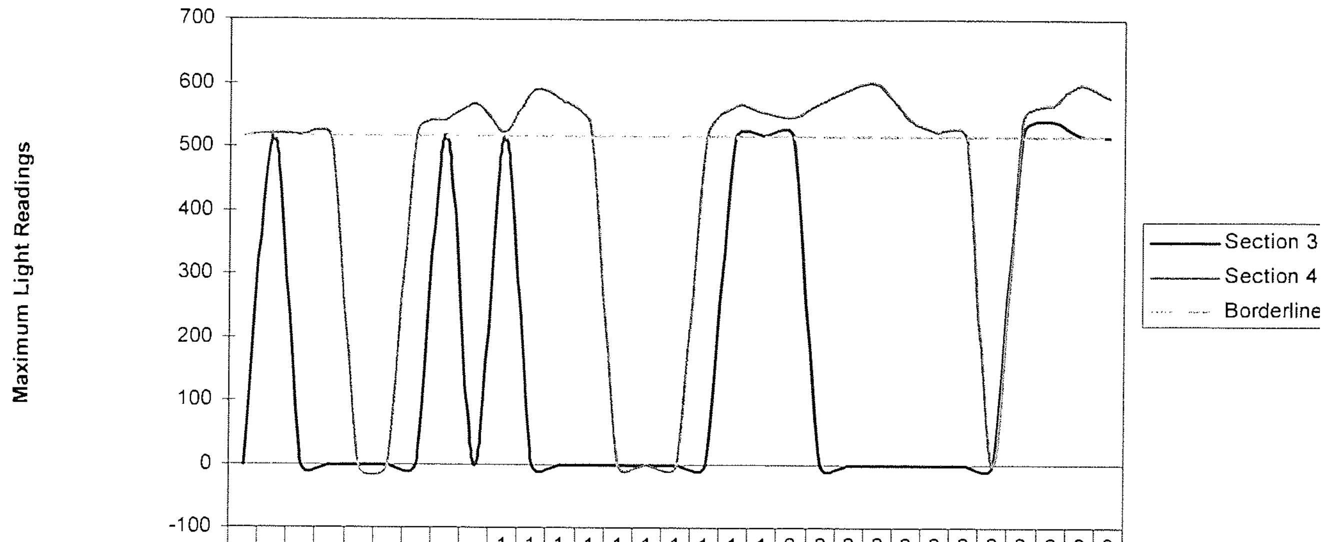
In order to obtain a critical parameter for the light readings we instructed the wireless sensor network nodes to report one light reading per packet for 2 minutes during the beginning of the trial. From the collected results from all 8 sections we used as critical parameter the value that was reported most often from all the sensors. This value was set to be 515. We were aiming through the light trials to identify the sections with the longest fluctuations and the highest values.

Figure 5-28 and 5-29 display the maximum light readings that were reported from Sections 1-4 from the facilitator node 1 during September.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 1	0	516	0	0	518	0	0	527	0	0	0	515	0	0	0	0	0	0	518	0	0	0	0	0	0	0	0	0	519	0	0
Section 2	515	0	516	0	517	0	0	0	0	0	515	0	0	515	0	0	518	0	0	555	0	0	0	0	562	0	0	514	0	515	0
Bordeline	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515	515

Figure 5-28: Section 1 and 2 maximum light readings during September



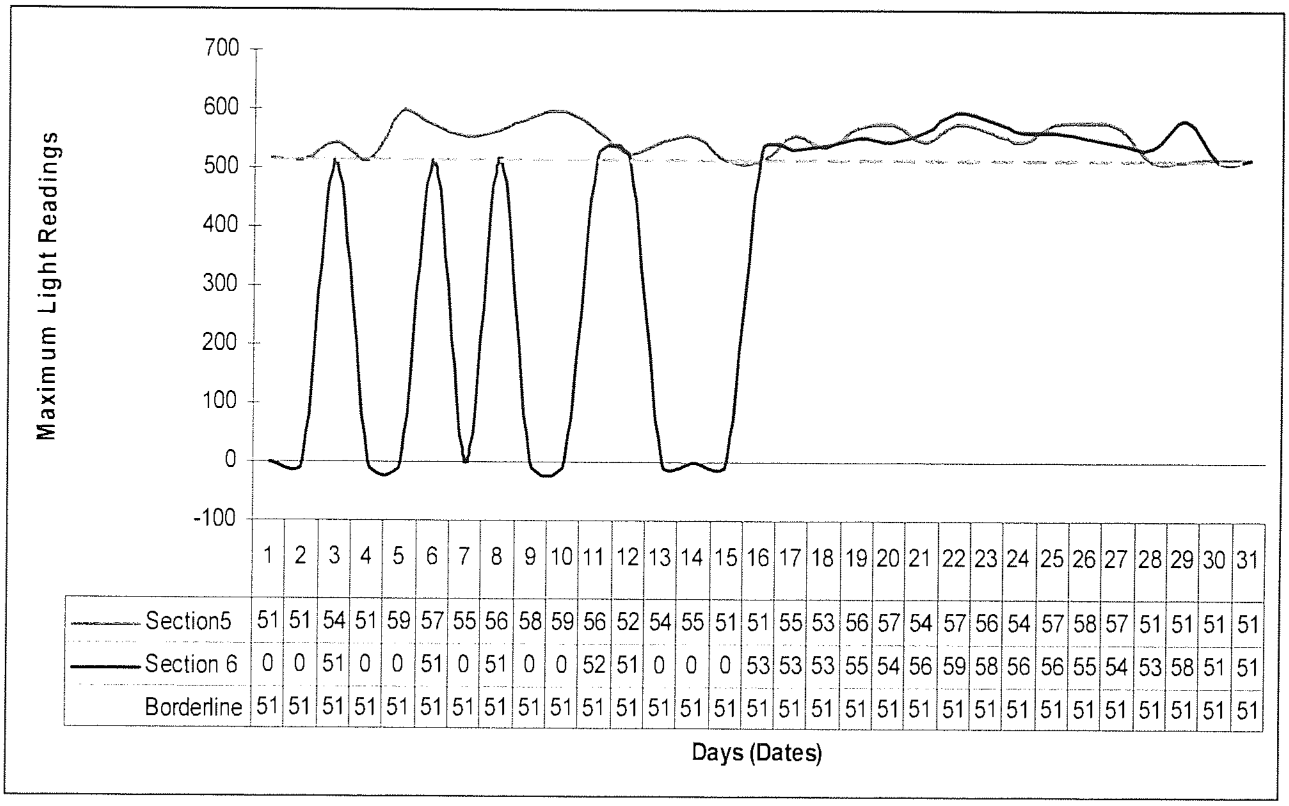
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 3	0	51	0	0	0	0	0	51	0	51	0	0	0	0	0	0	0	51	51	51	0	0	0	0	0	0	0	52	54	51	51
Section 4	51	52	51	51	0	0	52	54	56	52	58	57	53	0	0	0	51	56	55	54	57	59	59	54	52	51	0	54	56	59	57
Borderline	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51

Days (Dates)

Figure 5-29: Section 3 and 4 maximum light readings during September

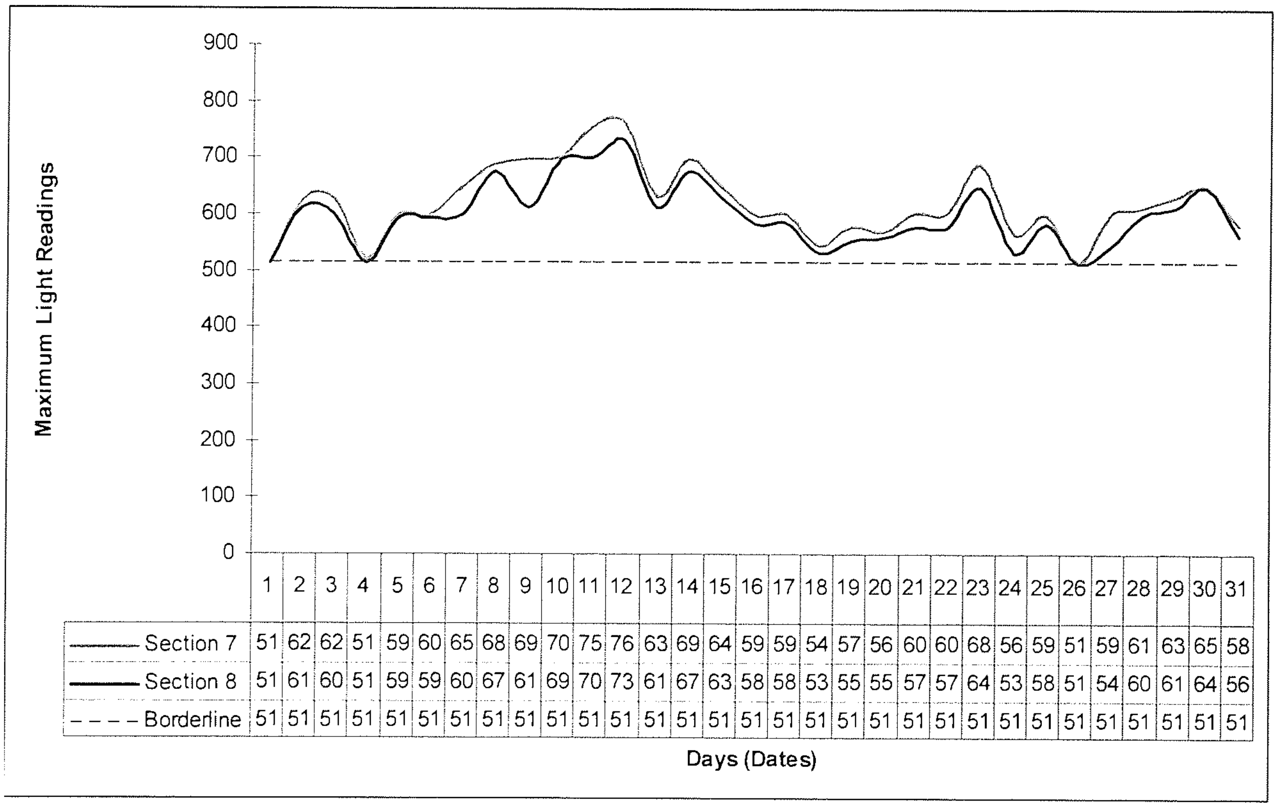
As can be seen from the above graphs Sections 1 and 2 where the ones that alerted the system the fewer times from the facilitator group one and they did not report frequent or very high light variations. This can be explained as they were quite far away from the front windows of the store and they had a constant artificial lighting. On the other hand, Section 4, which was quite close to the store windows, had alerted the system every single day reporting readings to the facilitator node 1 that exceeded quite often the borderline of 515.

Figure 5-30 and 5-31 present the maximum light readings that were reported from Sections 5-8 from the facilitator node 2 during September.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section5	51	51	54	51	59	57	55	56	58	59	56	52	54	55	51	51	55	53	56	57	54	57	56	54	57	58	57	51	51	51	51
Section 6	0	0	51	0	0	51	0	51	0	0	52	51	0	0	0	53	53	53	55	54	56	59	58	56	56	55	54	53	58	51	51
Borderline	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51

Figure 5-30: Section 5 and 6 maximum light readings during September



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Section 7	51	62	62	51	59	60	65	68	69	70	75	76	63	69	64	59	59	54	57	56	60	60	68	56	59	51	59	61	63	65	58
Section 8	51	61	60	51	59	59	60	67	61	69	70	73	61	67	63	58	58	53	55	55	57	57	64	53	58	51	54	60	61	64	56
---- Borderline	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51	51

Figure 5-31: Section 7 and 8 maximum light readings during September

The facilitator at node 2 was by far more alerted than this of node 1. All of its Sections were relative close to the store windows affecting them accordingly. Thus, Section 7 and 8 which were the closest ones recorded the highest values and the most frequent light variations. Wines that were stored there and more precisely in Section 7 which the temperature analysis identified a major problem, were almost certain that if they stayed for a period of one week or more at the same location would be oxidized really fast.

5.5 The In-Motes Bins analysis

The two months we spend in the warehouse helped us to understand better the nature of the wireless sensor network we deployed and was the first test of our middleware in a real environment. Packet delivery did not affected by the distance between the aggregation point and the sensors rather from facts such as human interference with the environment and location in terms of eye of sight of the sensors as the below analysis presents.

The maximum number of packets that Section 1 should report back to the terminal according with the application was 21,600 per trial that was lasting for 12 hours per day, 1h and 20 minutes of which was constantly sending readings to the base station, 1 reading per second for a period of 1 minute. Figure 5-32 demonstrates the total number of packets, containing one reading each, that were received per day from Section 1 during July.

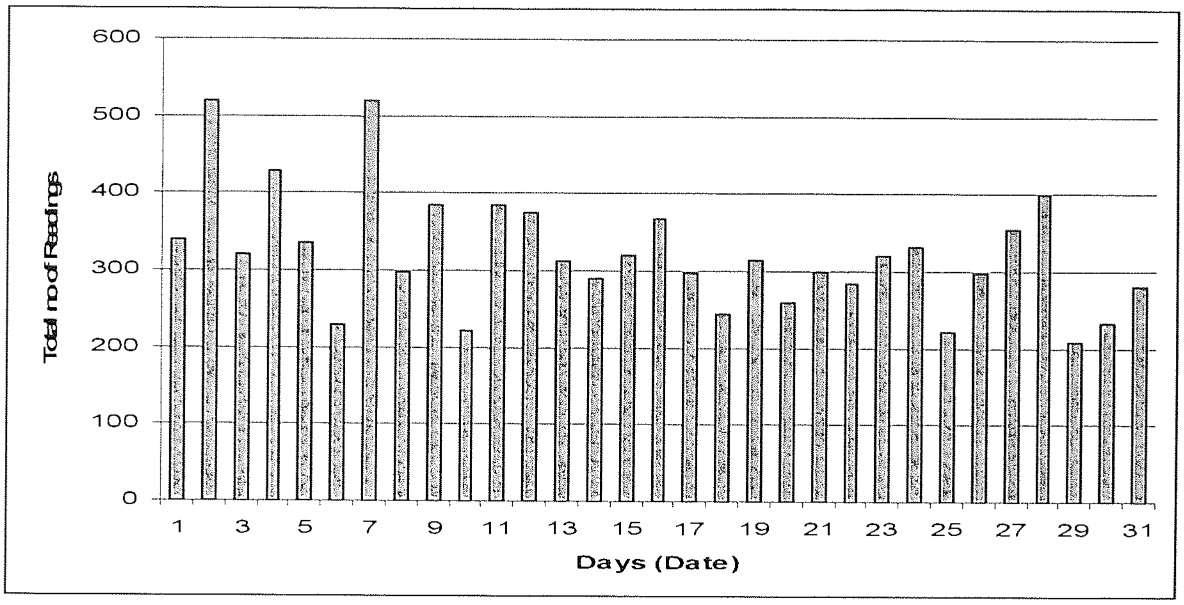


Figure 5-32: Total number of readings received from the base station from Section 1during July

It is obvious from the above graph that we didn't reach at any point the maximum number of readings per day. We concluded that the success rate of receiving the total readings from Section 1 was 1.49% during July and 1.55% during June.

Reasons behind this rate were memory limitations from the sensor and interference from the employees which were moving around the warehouse during the transmissions both resulting in the sensor stalling its operation. Even though, the sample of readings was large enough to provide us with a clear picture of the temperature variations of Section 1 during those months.

The total numbers of readings that Section 2 produced was slightly better than this of Section 1 during July, Figure 5-33.

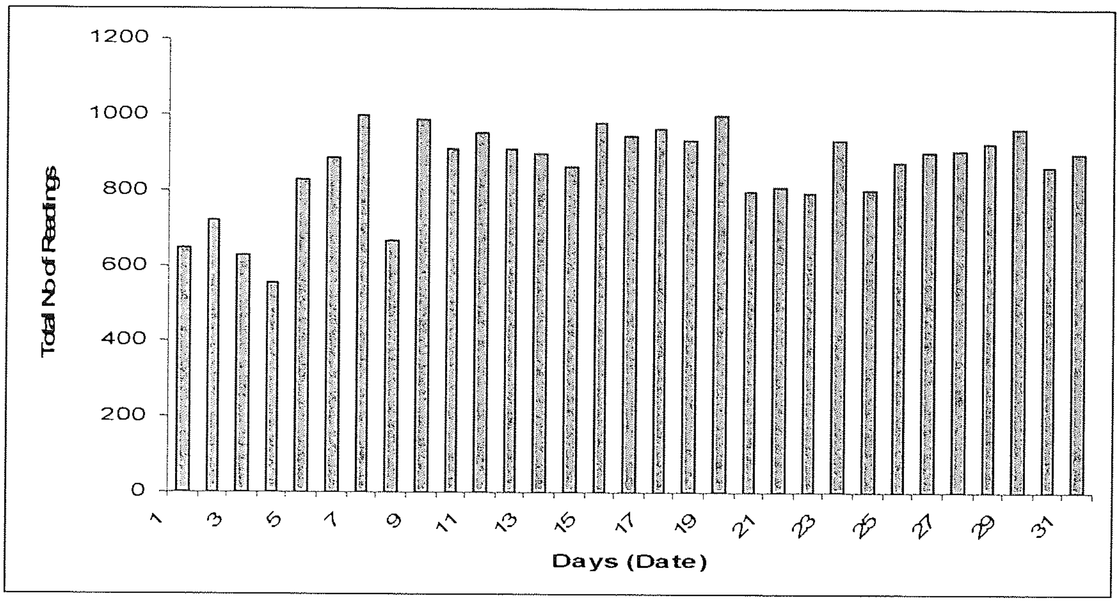


Figure 5-33: Total number of readings received from the base station from Section 2 during July

Although still below the maximum number of readings the sensor could send to the base station, the successful rate of packet delivery was 4% during June and 3.87% during July. This increase can be explained due to the fact that the area was not affected by the employee's movements during the transmissions in a same way we observed in Section 1.

Section 3 also had the worst rate of packets delivered to the base station, Figure 5-34.

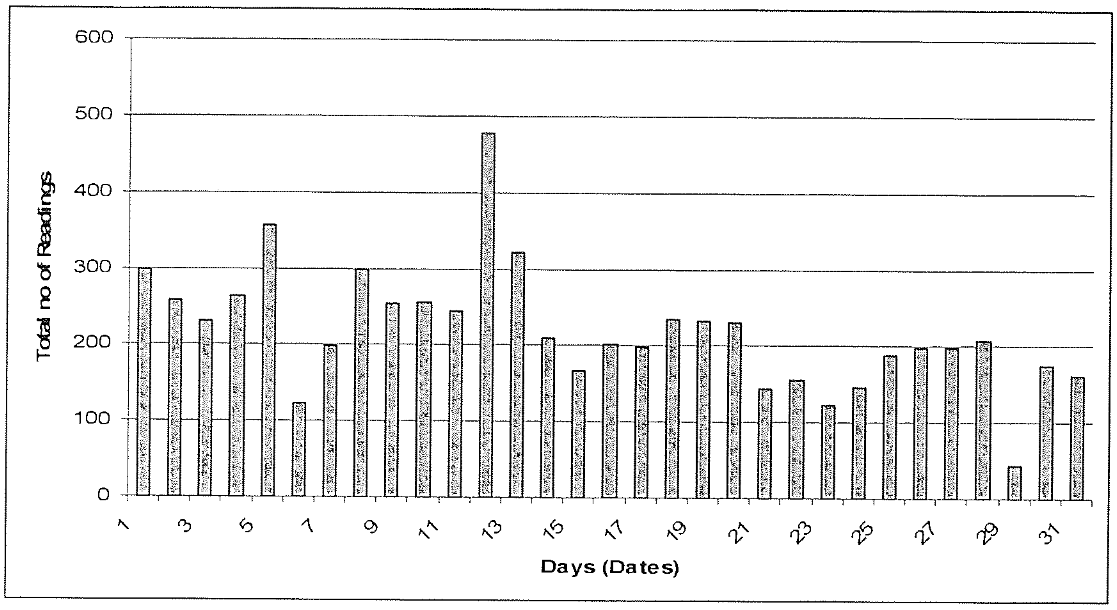
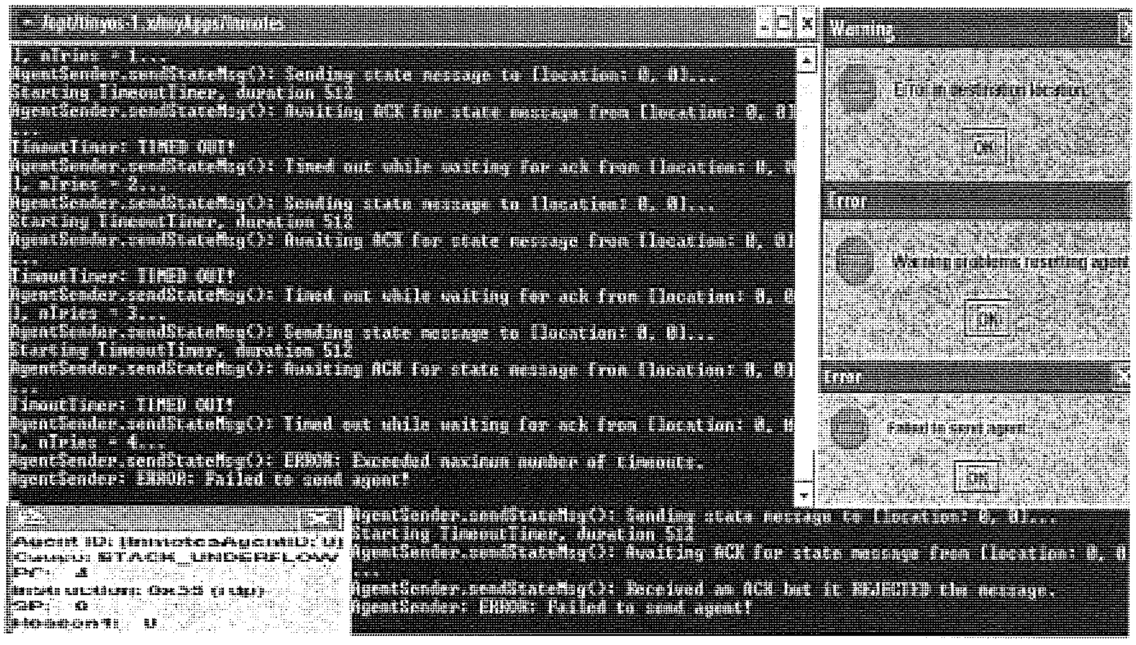


Figure 5-34: Total number of readings received from the base station from Section 3 during July

The successful rate was on the levels of 1.08% during June and 1.11% during July identifying that the transmission was affected by factors such as the busyness of the environment and of the fact that the sensor was not in the line of sight of the interface board of the aggregation point. Still, the data that was collected was sufficient enough for the purpose of the experiment.

Overall, we observed and recorded the following type of errors (Figure 5-35) for the duration of the warehouse trial:


```
[illegible]
I, nTries = 1...
AgentSender.sendStateMsg(): Sending state message to [location: 0, 0]...
Starting TimeoutTimer, duration 512
AgentSender.sendStateMsg(): Awaiting ACK for state message from [location: 0, 0]
...
TimeoutTimer: TIMED OUT!
AgentSender.sendStateMsg(): Timed out while waiting for ack from [location: 0, 0
I, nTries = 2...
AgentSender.sendStateMsg(): Sending state message to [location: 0, 0]...
Starting TimeoutTimer, duration 512
AgentSender.sendStateMsg(): Awaiting ACK for state message from [location: 0, 0]
...
TimeoutTimer: TIMED OUT!
AgentSender.sendStateMsg(): Timed out while waiting for ack from [location: 0, 0
I, nTries = 3...
AgentSender.sendStateMsg(): Sending state message to [location: 0, 0]...
Starting TimeoutTimer, duration 512
AgentSender.sendStateMsg(): Awaiting ACK for state message from [location: 0, 0]
...
TimeoutTimer: TIMED OUT!
AgentSender.sendStateMsg(): Timed out while waiting for ack from [location: 0, 0
I, nTries = 4...
AgentSender.sendStateMsg(): ERROR: Exceeded maximum number of timeouts.
AgentSender: ERROR: Failed to send agent!
```

Warning

Error in destination location

OK

Error

Warning problems inserting agent

OK

Error

Failed to send agent

OK

```
Agent ID: [InmotesAgentID: 0]
Cause: STACK_UNDERFLOW
PC: 4
Instruction: 0x35 (cup)
SP: 0
Heap cnt: 0
```

```
AgentSender.sendStateMsg(): Sending state message to [location: 0, 0]...
Starting TimeoutTimer, duration 512
AgentSender.sendStateMsg(): Awaiting ACK for state message from [location: 0, 0
...
AgentSender.sendStateMsg(): Received an ACK but it REJECTED the message.
AgentSender: ERROR: Failed to send agent!
```

Figure 5-35: A collection of In-Motes errors/warnings notifications as they were displayed during the trials

1. Stall of the whole network: 25

Describes the condition where the whole network was inactive and no readings were received at the user end. Reasons behind this behavior could be identified due to the below error types. The remote action that was taken was to send a fix In-Motes agent to flash the memory of all the network nodes and reinstall the application.

2. AgentSender Fail sending agent: 280

Describes the condition where one or more nodes were not responding to user requests. The mobile code transmitted from the facilitator node never reached its destination. The middleware was providing this information and the actions that were taken were: Either send a fix In-Motes agent to flash the node's memory and then send the user request again with a new In-Motes job agent or physically visit the node turn it off and on again. The problem was visualized from the user as the problematic node was blinking its red LED.

3. Null Readings: 350

Describes the condition where the facilitator node was sending back to the end user a null reading. No actions were taken place.

4. Facilitator Buffer_Overflow: 275

Describes the condition where the facilitator node could not handle all the receiving traffic resulting in stalling its operation. The red LED was blinking and a pop up window was informing the user about the error. A fix In-Motes agent was send from the user in order the internal memory of the node to be flashed.

5. Node Buffer_Overflow: 180

Describes the condition where one or more nodes of the network could not handle any queries and wasn't reporting back to the facilitator node although it had accepted and stored a new instruction (AgentSender Success sending agent). The red LED was blinking and a pop up window was informing the user about the error. A fix In-Motes agent was send from the user in order the internal memory of the node to be flashed.

6. AgentReceiver Fail receiving agent: 231

Describes the condition where one or more nodes of the network could not handle any queries and wasn't reporting back to the facilitator node. Although the destined node had accepted (AgentSender Success sending agent) the new In-Motes job agent the new instruction was never matched with any template in the tuplespace so the new tuple that was added did not trigger any reaction from the node. The red LED was blinking and the In-Motes engine was informing the user about the error. A fix In-Motes agent was send from the user in order the internal memory of the node to be flashed.

7. Misplacement of sensor/Drop/Other: 30

Describes the condition where a sensor accidentally was misplaced, dropped, or switched off during its operation.

From the above error codes and the behavior of the network during the warehouse trial we concluded that the main reasons of a mote to malfunction in a relative active physical environment were:

- The *traffic*, which in some nodes could easily lead to packet losses or store and forward delays for the facilitator nodes, packets containing either sensor results back to the end user or parts of an In-Motes agent. Re-transmission that were taken place incase of an In-Motes agent not reaching its destination did not decrease the levels of the problem.

- The *limited memory of the hardware* that did not allow us to store more instructions to the actual mote. The In-Motes engine virtual memory and the Instruction Manager although provide better memory management at that set up still did not lead to an ideal memory allocation.

- The *battery levels of a mote*. For every trial (warehouse/main floor) we were using a new set of AA batteries for every mote. We observed that the network under the same conditions and set up was producing more errors such as the ones mentioned above towards the end of the first month.

- The *physical interference*. The warehouse was a relative active environment with staff members moving in various locations during the trial. We observed that errors that recorded above tend to be more frequent when during a transmission from a specific location physical activity was taken place i.e. someone was moving and repositioning boxes in that area.

- The *reaction registry and the instruction manager of In-Motes*. The In-Motes communication protocol that was based on a tuplespace scheme with added reactions produced some minor problems, as we recorded that at certain instances although an In-Motes agent had arrived and stored a new tuple in the tuplespace of a mote, a reaction was never fired resulting the stall of the mote.

The In-Motes Bins trial in the main floor tested the merits of the middleware at a maximum level. The sensors at random time intervals continued to stall their operation and especially in Sections that were alerted more often. When a mote was offline or stalled its operation the rest of the network continued to function normally and we did not observe any catastrophic scenario where the network or a group of nodes would stop working due to this reason.

As before, the moment this error was noticed, simply we were sending a fix agent to that specific mote and we were flushing its memory. Due to the facilitator communication protocol that we presented in the previous chapter the slave node was re – captured and joined the same federation as before, a process that was accomplished in less than 60 sec from the moment the error was observed.

In terms of packet delivery, the performance rate was higher although we were using more motes in the wireless sensor network. Figure 5-36 presents the maximum number of temperature packets send from Section 7 node during August. We realized that the introduction of the critical parameter in the application had improved the dissemination of the available resources for the wireless sensor network. Thus, as an immediate effect the packet delivery performance of the network was improved even for sensors that they were active most of the time. The successful rate of packet delivery for Section 7 was calculated, according with equation 3, to be 25.6% during August and 36.3% during September. This performance did not get affected by factors such as the location or distance from the aggregation point or the topology of the nodes in the environment that was quite dynamic due to the fact that nodes accidentally could be misplaced from customers or staff.

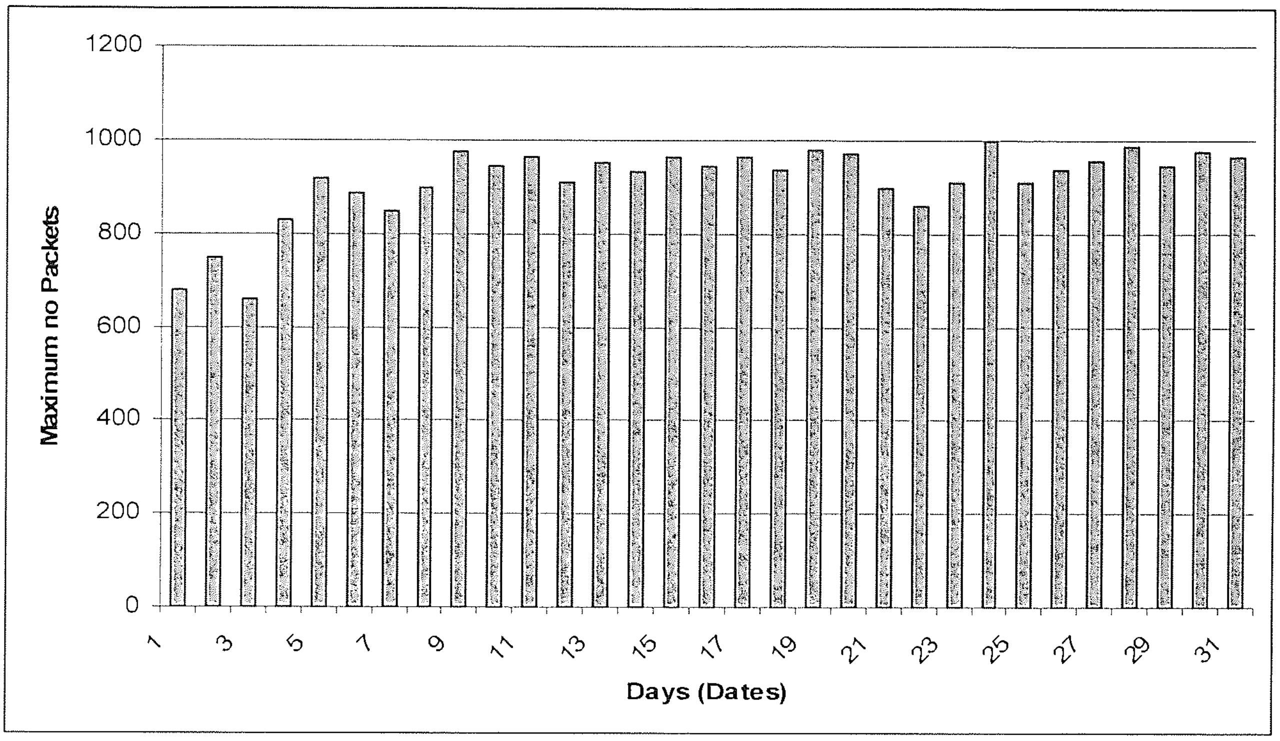


Figure 5-36: Total number of readings received from the base station from Section 7 during August

Also, we did not observe any store and forward delays, from the facilitator nodes as it was very rarely phenomenon all four nodes under the same facilitator to be activated at the same time. We applied some test also where one facilitator node will report only light readings while the second one only temperature ones and we did not observe any problems. The flexibility of the middleware was even allowing us to instruct individual nodes of the same facilitator to report dual readings at random interval and as before no transmission or packet delivery problems where observed.

Applying a WSN application in a dynamic environment such as that of the wine store can introduce many unexpected errors. People can accidentally misplace or drop a node and this can cause node or even network failures. To deal with these problems we introduced a very flexible interface where a job request or even the whole application could be reinstalled in less than 1min. Also our high level architecture based on the bacterial strains behavioural rules and the facilitator communication model enabled sections of the network to continue to report readings even if node failures were present (Georgoulas & Blow 2007).

Overall, we observed and recorded the following type of errors for the last two months of the main floor trial:

1. Stall of the whole network: 20
2. AgentSender Fail sending agent: 200
3. Null Readings: 400
4. Facilitator Buffer_Overflow: 200
5. Node Buffer_Overflow: 120
6. AgentReceiver Fail receiving agent: 245
7. Misplacement of sensor/Drop/Other: 40

As can be seen from the below graph (Figure 5-37) both the middleware and the network produced less errors although the scale of the application was larger. The In-Motes engine and all the middleware functions including the communication protocol were unmodified. The only changes that took place, as we discussed earlier, were the dynamic In-Motes job agents and the insertion of another facilitator node. Thus, the creation of a more flexible application reduced both the traffic and the energy levels of the individual motes resulting in less network errors and better packet delivery.

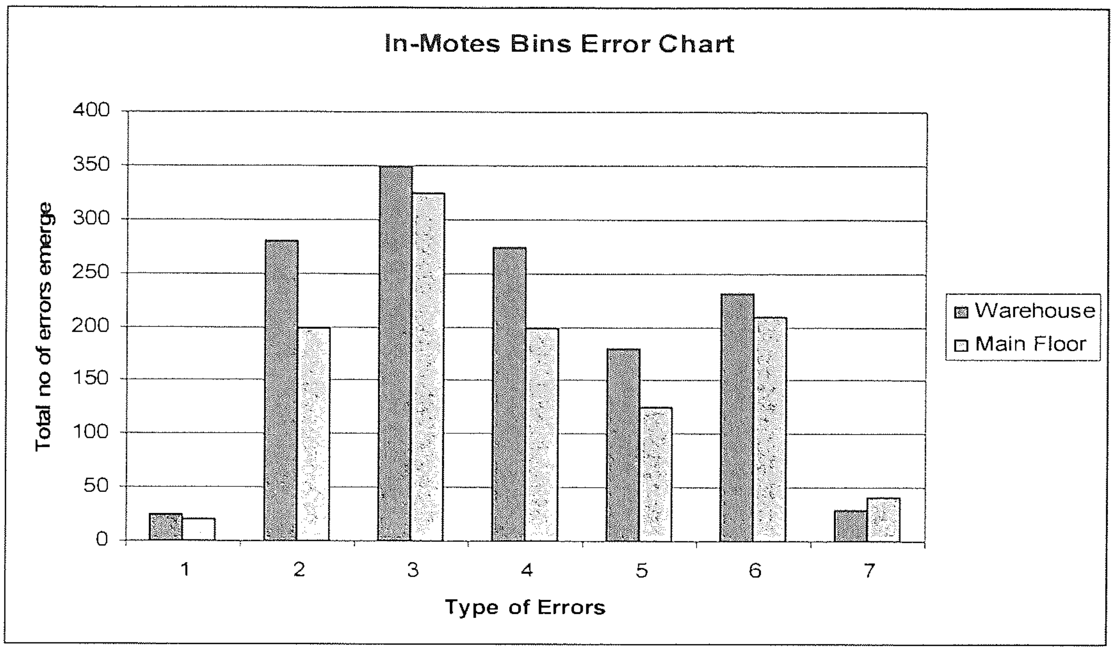


Figure 5-37: Comparison error chart between the warehouse and the main floor trials

5.6 Summary

This chapter presents the In-Motes V1.0 middleware testing and evaluation. Initially we carry out experiment in the controlled environment of the labs where we observed the behaviour both of the wireless sensor network and the middleware. We compared and analysed our approach with two middlewares by creating a simple application, in order to see where our middleware stands in comparison with the other two approaches. Having identified a number of problems through those trials such as frequent stall of the motes and network failures we wanted to test our middleware in a real environment. Therefore, we created the In-Motes Bins application that was deployed in two phases for a period of four months in a wine merchant chain store. We were able through our trial to provide with a detailed analysis of the environmental conditions of the store and identified bad sections which were causing problems to the selling product. Graphs were produced showing temperature and light variation in both the warehouse and the main floor of the store. The merits of our middleware were also tested and various conclusions were made. For example, through the packet delivery analysis we observed that the transmission of packets is highly dependent with the battery power of the motes and concluded that the more a mote is used in an environment the more susceptible would be to failures such as stalls of its operation as its battery levels are reduced.

CHAPTER 6

The In-Motes Reloaded Analysis

6.1 Introduction

The aim of this chapter is to present the In-Motes Reloaded middleware and the application that was developed, as part of the testing of the updated platform, called In-Motes EYE. The four months trial in the wine store of the In-Motes through the In-Motes Bins application offered us a lot of valuable lessons concerned with the applicability and adaptability both of the wireless sensor network and of the actual middleware we developed. The next step was to use this experience and produce an updated version of In-Motes which we called In-Motes Reloaded. In the next sections we are going to present the number of changes that were applied in some of the core functions of In-Motes together with additional attributes that were added in an effort to make our middleware more adaptable and flexible than before. In order to test the merits of the In-Motes Reloaded we developed an application that was measuring the acceleration of moving objects in an environment and which we named it In-Motes EYE.

6.2 The In-Motes Reloaded middleware design

With the updated version of the middleware design we want to address certain issues such as the fault tolerance of the software and its ability to reconfigure effectively the wireless sensor network. Also we realized through the previous trials that if motes are overactive in an environment they will quite often stall their operation. The reasons, behind that behaviour as was shown in the previous chapter, were mainly the memory limitations of the motes and the energy consumption of the batteries. Thus, with In-Motes Reloaded our basic aim is to overcome those problems by creating a more flexible and less energy consuming platform for the applications that we will deploy in the wireless sensor network. As we are going to see in the next paragraphs we focus on altering some of the In-Motes engine parameters, add more op codes and nesC interfaces, update the instruction set and in general pass more responsibilities to the actual middleware, allowing us to deploy less complicated and less energy consuming applications to the wireless sensor network.

One of the foundational changes that can be found in the updated version of the middleware is the way in which the mobile code is transmitted inside the wireless sensor network. As we showed, with the In-Motes Bins application, this attribute was application dependent and what we name In-Motes agents could travel either by migrating or cloning. In-Motes Reloaded specifies that In-Motes agents are able to travel from one node to another only by migrating. The cloning instruction and the according op code still exist in the middleware, incase a developer wants to use it, but unless it is instructed otherwise through the deployed application the middleware assumes and registers all In-Motes agents for migration for all the applications. Thus, there is no need to add a cloning or migrating instruction to the application, as we did before upon deployment, as the rule manager of the middleware will deal with it. Figure 6-1, presents part of the nesC interface code, AgentMgr.nc, for the In-Motes agent migration that is called inside the rule manager of the In-Motes Reloaded, the commented section is part of the cloning method that we do not use anymore.

```
      // Some code is missing…
if (call LocationUtilI.equals(&dest, call ContextMgrI.getLoc())) {
      // Allow an agent to migrate
        (op == OPmove) {
        context->condition = 1; // move was successful
        context->pc = 0; // reset program counter
        call OpStackI.reset(context);
        call HeapMgrI.reset(context);
        call RxnMgrI.flush(&context->id);
        return SUCCESS;
      }
    }
            // WARNING: IF YOU UNCOMMENT THE FOLLOWING LINES YOU NEED
            //TO SPECIFY IN THE APPLICATION IF YOUR AGENTS ARE
            //MIGRATING OR CLONING
            //if (op == OPclone) {
            //id.id = call AgentMgrI.getNewID();   // create a new ID
            //for the clone
      //Some code is missing…
```

Figure 6-1: The above nesC code was allowing all the In-Motes Reloaded agents to migrate in order to move inside the wireless sensor network

Having used the cloning method in the warehouse of the store for transmitting agents we realized that this method was quite consuming both in terms of memory and energy. By flooding the available network with clones of a single In-Motes job agent means that nodes are awake more often therefore consume more energy for every transmission. Also, as we have mentioned in earlier chapters, the agents of our middleware are transmitted not as a single packet but they are divided into a number of packets in an effort to overcome the unreliable bandwidth of the wireless sensor network. Thus, cloning was increasing the traffic inside the network and the number of re-transmissions from every node, incase of a packet loss, affecting badly both the middleware as well as the nodes memory/energy resources.

The In-Motes Reloaded engine is updated as well, through the InMotes.h component in an effort to create a better debugging option for the developer and a better management scheme of the available memory of the system in an effort to increase the fault tolerance of the middleware's core engine.

Figure 6-2 presents the additional debugging code that is added to the In-Motes Reloaded engine interface. The main reason behind this addition is to identify, before we install and run any applications, the correct functionalities of the facilitator and slave In-Motes agents and acknowledge any initial errors such as race conditions and queue overflows.

```
WARNING: COMMENT THE FOLLOWING LINES WHEN YOU ARE RUNNING AN
APPLICATION
#define EXPERIMENT_MODE_INMOTES_RELOADED
#define NUM_EXP_EPOCH 4
#define BASESTATION_ADDR 0
#define UART_X 0
#define UART_Y 1
#define BCAST_X 0
#define BCAST_Y 2
// epoch settings
#define EPOCH_PERIOD    2048 // base epoch rate
#define INMOTES_RAND      1024   // random delay
#define INMOTES_TIMEOUT   10240 // time before visiting a neighbour
// The FACILITATOR can communicate with 4 SLAVE MOTES
#define INMOTES_MAX_NUM_SLAVES 4 // must = MAX(4)
// blinks the red and green LEDs for half a second when the
//facilitator captures and communicates with the slaves
#define SHOW_BOUNCE_LED
#define BOUND_LED_TIME 256
#define REPORT_ALL_ERRORS
```

Figure 6-2: The additional experimental code that was added to the In-Motes Reloaded engine for identifying errors prior the deployment of an application

The above code, when uncommented, is allowing the developer prior of the deployment of any application to test the communication protocol and the capturing procedure of the facilitator agents, verifying in parallel that all the available nodes are communicating with the right facilitator. Also, in the end of the predefined epoch it produces a list of any initial errors during this communication set up that otherwise would have passed undetected.

The engine is also initialized with a set of new values for important parameters such as the number of In-Motes agents available in a node per measuring period and the size of the operand stack of the middleware. In the first version of the In-Motes engine, a node could handle up to 5 different In-Motes agents at the same time. During the In-Motes Bins application this number was reduced to 2 and now In-Motes Reloaded is assigning only 1 In-Motes agent per node in the wireless sensor network. Reasons behind these reductions were purely memory limitations of the sensor devices caused by the number of instructions that were processed per agent during an application. Also, by having a large number of mobile codes residing in a node but without been active was an extra cost for the virtual memory of the middleware. Using dynamic In-Motes agents, as we did in the case of the In-Motes Bins application, a single In-Motes agent per node was sufficient for the execution of even complex scenarios.

The above lightweight approach of assigning a single In-Motes agent per node allowed us to reduce the size of the operand stack in bytes, since the numbers of instructions that would be processed by a node were less than before. Thus, the In-Motes Reloaded operand stuck is set to be 100 bytes from 150 bytes that it used to be. This change improves the middleware's ability to handle multiple packages and queries by allocating more resources to the In-Motes Reloaded Rules manager and Facilitator Manager.

Last but not least, the states of an agent inside the wireless sensor network are updated as well, Figure 6-3. To the already existed ones of RUN, SLEEP and WAITING we add the ARRIVING, LEAVING, HALT, READY and BLOCKED. Thus, we are able to control better the interconnections between the In-Motes agents and we can identify easier any deadlocks during the lifetime of an agent, caused mainly by packet losses during a transmission.

```
  typedef enum {
  INMOTES_STATE_HALT          = 0, // no agent present in a node
  INMOTES_STATE_WAITING       = 1,
  INMOTES_STATE_RUN           = 2, // execute the given reaction
  INMOTES_STATE_ARRIVING      = 3,
  INMOTES_STATE_LEAVING       = 4,
  INMOTES_STATE_BLOCKED       = 5, // due to behavioural rules
  INMOTES_STATE_READY         = 6,
  INMOTES_STATE_SLEEPING      = 7, // sleeping waiting for a reaction
  INMOTES_STATE_WAIT          = 8,
} InmotesAgentContextState;
```

Figure 6-3: The In-Motes Reloaded agent states were updated by adding more in an effort to control better their interconnections

Based on the above modifications the MigrationMsgs.h nesC module was updated. Thus, the In-Motes Reloaded sender buffer size is reduced to 2 from 4 and the receiving buffer size from 3 to 2. By this reduction we are aiming to overcome the buffer overflow problems that we faced when we had more agents in the sending and transmitting end, Figure 6-4.

```
enum {
  INMOTES_HEAP_MSG_SIZE = 32,
  INMOTES_OS_MSG_SIZE   = 31,
  INMOTES_SNDR_BUFF_SIZE = 2, // max number of agents being
                              //sent
  INMOTES_RCVR_BUFF_SIZE = 2, // max number of agents arriving
  INMOTES_SNDR_MAX_TIMEOUTS = 4,
  INMOTES_SNDR_MAX_RETRIES = 2,
```

Figure 6-4: The In-Motes Reloaded Sender/Receiver buffer changes were aiming to overcome the buffer overflow problem

Modifications are also applied in the tuple space specification in an effort to conserve energy during the communication of an In-Motes job agent with the hardware and also to overcome the error we noticed in the In-Motes Bins application with specific reactions not been able to trigger a predefined action from the node.

Thus, we alter the TupleSpace.h interface by decreasing both the individual size of the tuple spaces allocated physical memory and the maximum number of field types for every tuple, Figure 6-5.

```
typedef enum {
  INMOTES_TS_SIZE = 50,      // size of RAM tuple space in bytes
  INMOTES_MAX_TUPLE_SIZE = 25, // the max number of field bytes in a
                               //tuple
  INMOTES_OUT_Q_SIZE = 3, // the max number of pending OUT operations
  INMOTES_RTS_TIMEOUT = 512 // remote op timeout in binary
                        // milliseconds
  INMOTES_RTS_MAX_NUM_TRIES = 2   // the max number of times a
                                    //request is sent
  REACTION_MGR_BUFFER_SIZE = 5 // the max number of reactions
} TupleSpaceIConstants;
```

Figure 6-5: The In-Motes Reloaded tuple space modifications

The In-Motes Reloaded tuple space size is reduced by 50 bytes and the maximum number of field types per tuple by 25. With these changes we are aiming to reduce the physical memory that every mote is consuming when a reaction enters a tuple space. Moreover, we left the maximum number of reactions a tuple can handle the same but we decreased the maximum number of request we were sending in order to minimize any chances for race conditions to appear during a transmission that would lead the mote to stall its operation.

The In-Motes Reloaded middleware comes with a new graphical user interface, Appendix B, which allows the user to deploy applications to the wireless sensor network easier and faster. Thus, agent definitions such as fix agents and facilitator agents are loaded to the wireless sensor network directly with the press of a button since their code exists in the middlewares engine. Also, the In-Motes Bins application can be selected directly as well without the user to have to write any code for it. A grid format for the wireless sensor network is also available, as described in the In-Motes Reloaded User Guide in Appendix B. We used this format for debugging reasons only but it can be applied to certain applications as well, allowing the user to instruct individually every node with a specific unique task.

6.3 The preliminary tests of the In-Motes Reloaded

The SenseLight application that was developed for the preliminary tests for the In-Motes middleware was used to test the merits in terms of packet delivery performance of the In-Motes reloaded as well. The wireless sensor network this time consisted of 11 mica2 nodes and 1 MIB510 interface board. Two facilitator In-Motes agents were deployed at once and captured four slave nodes establishing the communication protocol prior to installing the application. The application was using 45 bytes job agents as we described before and were reporting one light reading per message for a period of two minutes. The longest period that the application was set up running was as before 10 minutes during which the packet delivery performance was varying. No catastrophic scenarios in terms of total network failure were observed. On average, after the first 100 seconds the decrease in performance was sustained and remained roughly constant on every trial even for the longest periods of continuous transmission (10 minutes).

Figure 6-6, presents the packet delivery performance of the In-Motes Reloaded when running the SenseLight application.

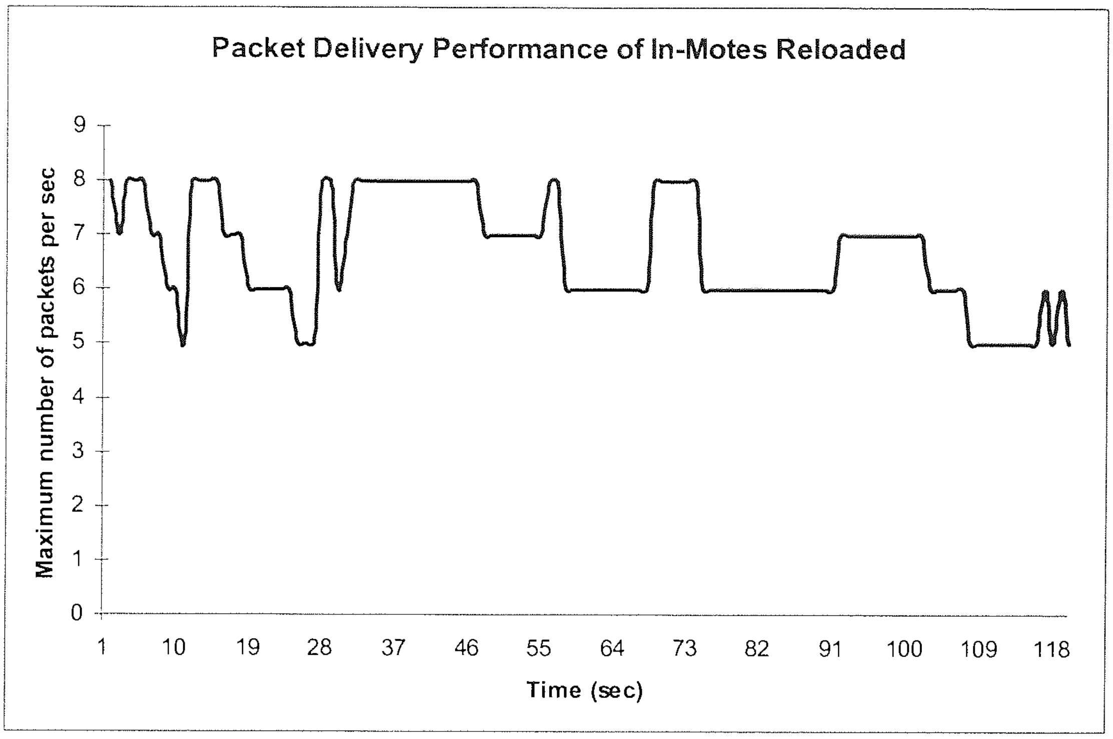


Figure 6-6: Packet Delivery Performance of In-Motes Reloaded over a period of 2 minutes

As can be seen from the above graph there were variations in the maximum number of packets that the middleware was reporting through its two facilitator nodes to the end user, having as the ideal value the eight packets per second. We concluded that the success rate of receiving the total number packets was 82.6%.

We run the SenseLight application with the same wireless sensor network specification using the In-Motes engine for a period of two minutes, Figure 6-7. The success rate of packets delivered this time was calculated to be 55.6%.

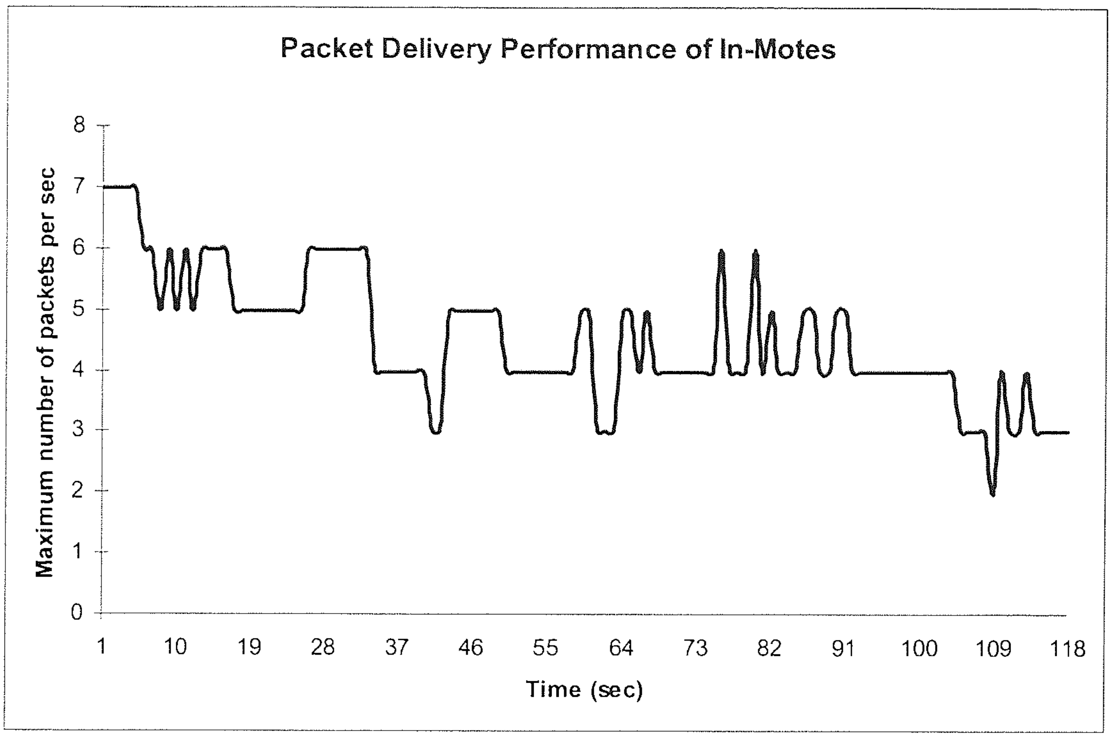


Figure 6-7: Packet Delivery Performance of In-Motes over a period of 2 minutes

We conclude that the modifications of the In-Motes engine has significantly improve the performance of the middleware, with the In-Motes Reloaded having a higher average packet throughput when compared with this of In-Motes, Figure 6-8. A point of concern is that the performance seems once again to be affected as time pass by with nodes reporting fewer packets to the end user. The reasons for this behaviour as before were mainly the memory limitations from the hardware especially in a scenario that was stating that a node would report continuously readings to the end user for a long period of time. The longest period that the application was set up running was 10 minutes during which the packet delivery performance was varying in a similar way with the displayed graph above, Figure 6-6. No catastrophic scenarios in terms of total network failure were observed. On average, after the first 100 seconds the decrease in performance was sustained and remained roughly constant on every trial.

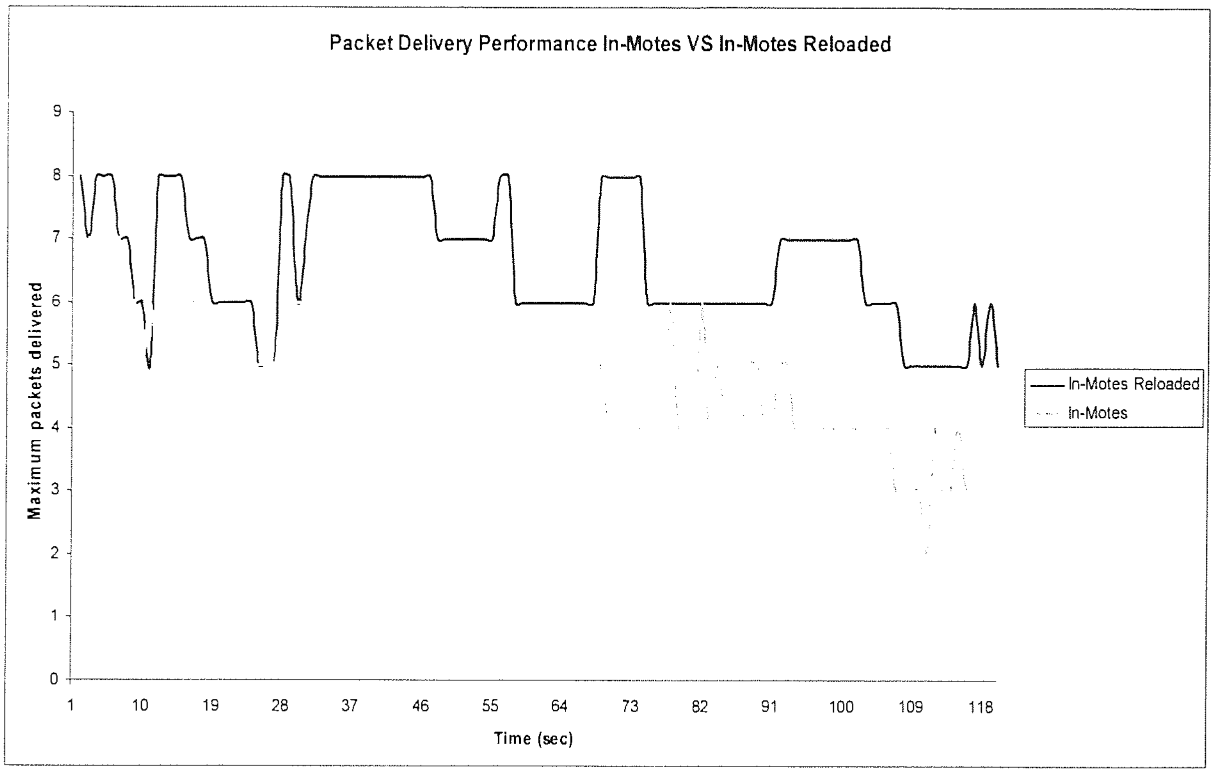


Figure 6-8: Packet Delivery Performance of In-Motes Reloaded and In-Motes over a period of two minutes

The updated engine of the middleware, the way that the mobile code was transmitted inside the network and what we described as dynamic behaviour of the motes were the three main factors that lead to the improved performance of the In-Motes Reloaded over In-Motes middleware. Even though we managed an increase in the throughput the problem of lower performance after some time of continuous transmissions still existed. As we explained before, the nature of the hardware (memory limitations, unreliable radio, battery levels) together with the large amounts of traffic during the longest trials created problems in the performance which was still decreased after some time.

6.4 The In-Motes EYE application

Having finished the preliminary tests of the In-Motes Reloaded middleware we wanted to test the merits of our software by deploying a more complicated application in terms of In-Motes agent interconnections and readings collection, in the wireless sensor network. Also, we wanted to verify if the packet delivery performance which we observed earlier would be established again in a more complex scenario. Last but not least the deployment of such an application would reflect on how well the In-Motes Reloaded middleware was performing when hurdles such as buffer overflows, race conditions and stalls of the hardware might appear. Thus, we developed the In-Motes EYE application, an agent based real time application for monitoring the acceleration of moving objects.

In terms of hardware we have used a set of 5 mica2 sensors with the accompanied MTS310CA sensor boards. One base station, a laptop connected with an MIB510 interface board fixed in an area served as the aggregation point. Two radio controlled cars with an attached mica2 sensor had the role of the moving objects in the environment, Figure 6-9. The last 2 mica2 sensors were occupied by the two facilitators and they where placed in a straight line 2 meters apart communicating with the aggregation point that was 6 meters away and in the line of sight the facilitator nodes. All the hardware was provided by the Crossbow Technology Ltd. All the sensor motes were working under the TinyOS operating system and the application was deployed through our In-Motes Reloaded middleware.

Our trial took place in an outdoor environment with a sufficient space for the radio controlled cars to accelerate without any obstacles. The laboratory controlled environment was avoided all together mainly because of the limited space and our assumption of noise interference from the laboratory equipments that seemed to interfere with the wireless sensor network transmissions.

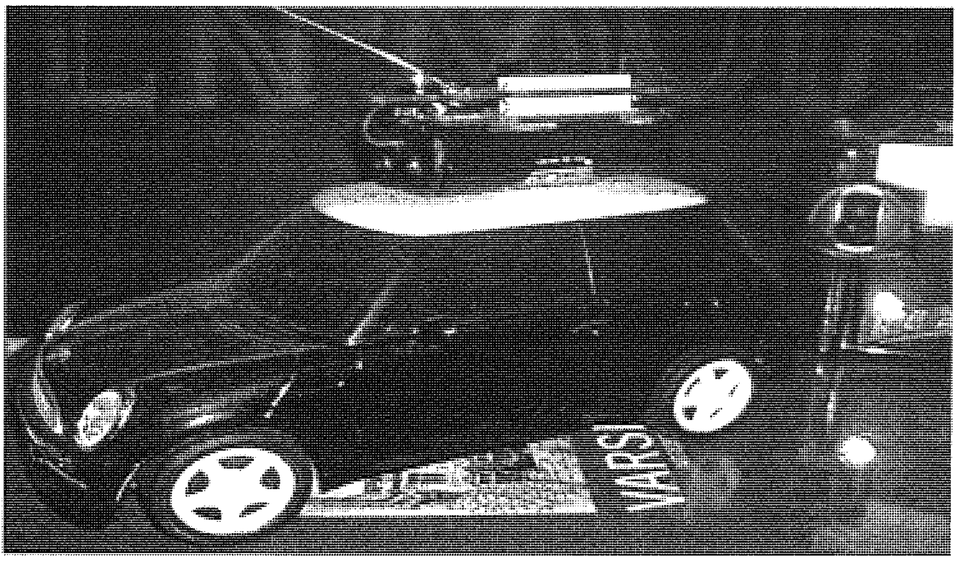


Figure 6-9: One of the radio controlled cars with the attached mica2 sensor that was used for the In-Motes EYE trial

The In-Motes EYE application uses dynamic job In-Motes agents for forwarding all the user queries in the wireless sensor network. Every monitor request that is send from the user contains a critical parameter which is used locally to determine if a sequence of readings should be reported or not. The nodes that are attached to the cars are waking up every two minutes and report to their facilitator node one acceleration reading. When the critical parameter is breached, each facilitator sends one packet containing one acceleration reading per 10 seconds for a period of 2 minutes.

The user also has the freedom to choose a random radio control car and at random intervals to check its acceleration behavior for a period of two minutes. In order to do so, the memory of the node of the selected car firstly must be flushed, by sending a fix In-Motes agent in order the resident job agent with the critical parameter and the according reactions to be erased. Then a new job agent is migrating to the desired location. After the end of the measuring period the job agent stops its execution and dies.

Figure 6-10, demonstrates part of the In-Motes EYE application code that was deployed to the wireless sensor network during our trials.

```
// Variable Declaration code omitted
      spushm ACCELX
      spushm CRT
      sense          // sense temperature or light
      IF          ACCELX > CRT THEN
      spushm 2
      spushloc
      rout 10        // remote out tuple containing acceleration
                  // reading to facilitator per 10 seconds
      spushm 1
      sleep // for 2 minutes
      ELSE
      sleep // for 2 minutes
      // some code is missing ...
      rstart BEGIN
```

Figure 6-10: Part of the In-Motes EYE application that was deployed from In-Motes Reloaded middleware

We initialize our wireless sensor network by injecting once two facilitator agents to the according nodes that are placed in a straight line and 2 meters apart, with one node attached to each car acting as slave to them. The facilitators captured the nodes and were ready to receive the first job requests in 35s. We send, once, two dynamic job agents, 40 bytes each to the wireless sensor network.

The above procedures, as well as the wake up calls and the transmission of data readings are executed by the application without us interfering with the process unless a failure is noticed or as we mentioned above a user desires to monitor the acceleration of a specific car.

6.5 The In-Motes EYE field tests

The main location of the In-Motes EYE trial was based on the outdoor environment of a garden. As we mentioned above the two facilitator nodes where placed in a straight line 2 meters apart and they were communicating with the aggregation point that was 6 meters away and in the line of sight the facilitator nodes, no physical obstacles where intervening during the transmissions. We used two radio controlled cars which at random time intervals were accelerating in a square area of the garden, 10x30 meters. The motion of the cars was random and it was not following any patterns, Figure 6-11.

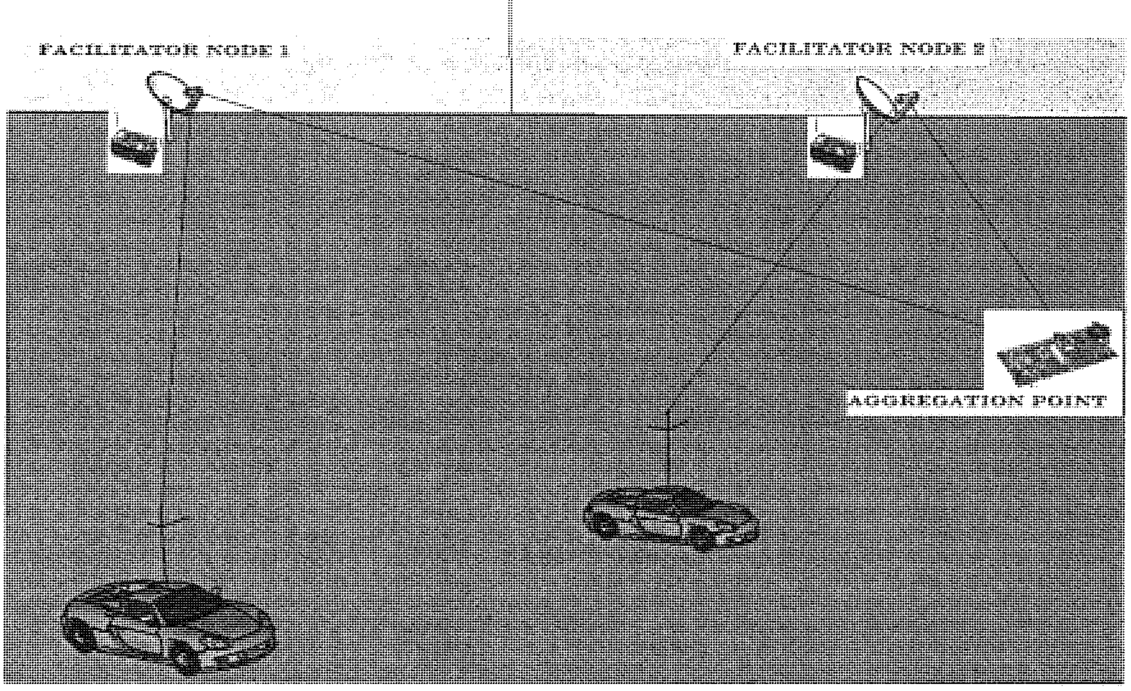


Figure 6-11: A representation of the In-Motes EYE trial environment and its actors

We used the accelerometer sensor which exists on the MTS310CA interface boards, a MEMS surface micro-machined 2-axis, ± 2 g device that can be used for tilt detection, movement, vibration, and/or seismic measurement (Crossbow User Guide

2004), Figure 6-12. According with the manufacturer (Analog Devices manual 2003) it is advised that for accurate measurements, after every trial the accelerometer needs to be recalibrated for every sensor in both axes. During our trials no recalibration of the devices took places as it was beyond the scope of the experimental procedure.

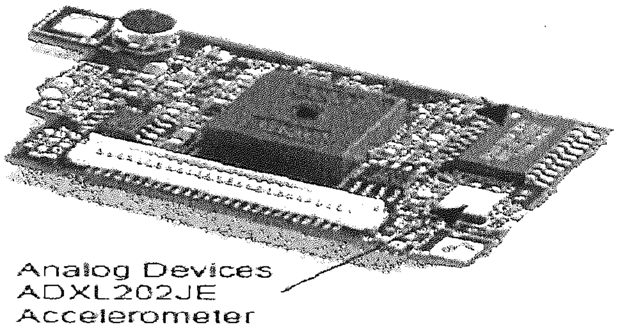


Figure 6-12: The accelerometer we used for the In-Motes EYE trials that was mounted on the MTS310CA interface boards

Since the voltage response for the accelerometer is linear with respect to the measured acceleration the motes ADC value can be translated into meaningful engineering acceleration units following the below linear equation:

$$\text{Reading } (m/s^2) = 1.0\ (\text{Cal\_pos\_1g} - \text{ADC})/\text{Scale factor} \quad (1)$$

Where:

Scale factor = Cal_pos_1g - Cal_neg_1g / 2

Cal_pos_1g = 500

Cal_neg_1g = 400

ADC = output value from Mote's ADC measurement

The critical parameter for the In-Motes EYE application was set to be 1.082 m/s^2, a value that was selected as it was the average acceleration reading reported from both of the radio controlled cars when they moved randomly in the environment.

Figure 6-13 is a representative graph that was produced when one of the radio controlled cars was accelerating above the critical parameter. The facilitator node that was assigned to that vehicle was reporting readings back to the end user for a period of 2 minutes by sending 1 packet containing one acceleration value per 10 seconds. Figure 6-13 does not represent continuous values of acceleration rather than values of acceleration when the critical parameter was breached, thus a single reading is reported that it only goes back to the previous transmission which happens every 10 seconds. Many of the graphs were produced by the In-Motes Reloaded Oscilloscope application allowing us to observe the acceleration of the cars in real time.

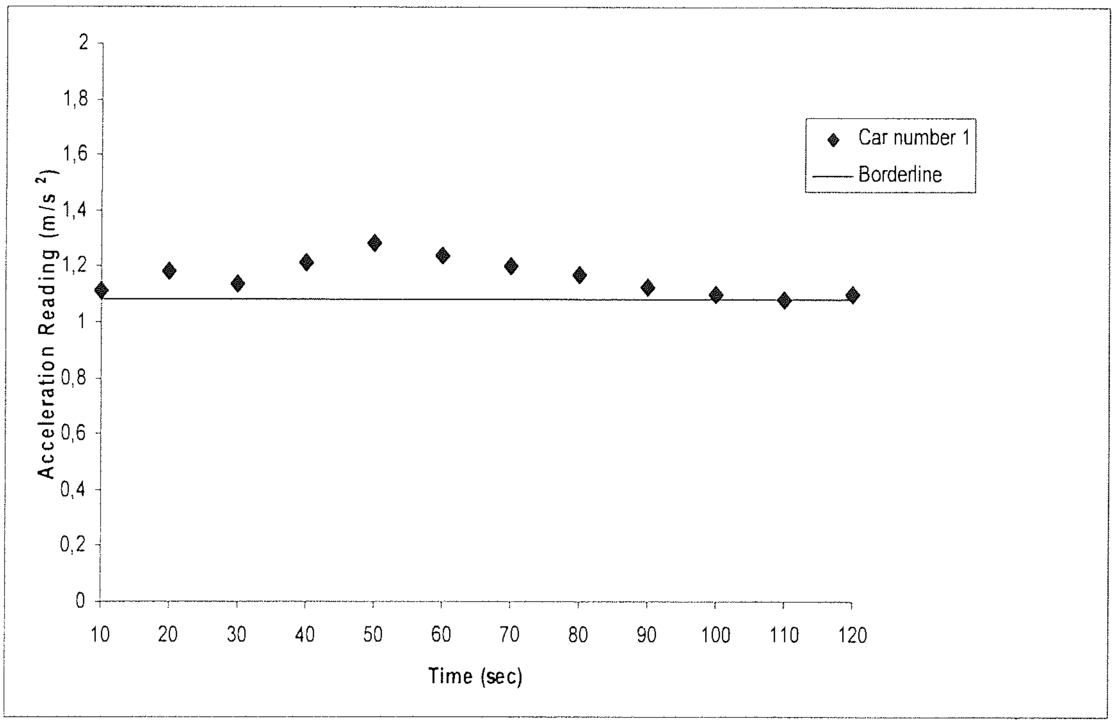


Figure 6-13: Car 1 acceleration readings reported to the end user when the critical parameter was breached

As we mentioned earlier the application was allowing a user to monitor the acceleration pattern of a moving car even if the critical parameter was not breached by simply injecting a new job agent to the vehicles sensor. Figure 6-14 presents the graph that was produced in the scenario where the user was monitoring the acceleration pattern of car 1 for a period of two minutes while car 2 was accelerating at the same time breaching the critical parameter of the application. The values for car 1 were the current single values of acceleration that were recorded every 10 seconds for the given period while the values for car 2 are as before values of acceleration when the critical parameter was breached. The sensors for each car in this case were working under two different dynamic job agents.

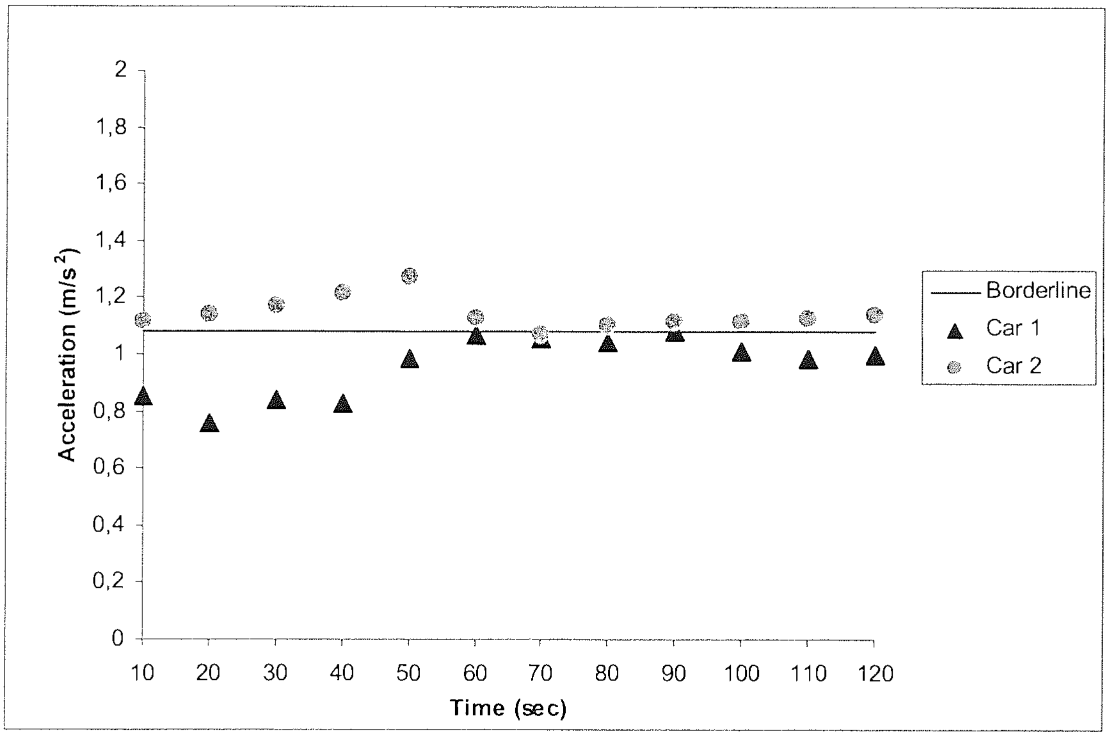


Figure 6-14: Car 1 is under surveillance by the user while Car 2 accelerates breaching the critical parameter

6.6 The In-Motes EYE analysis

Overall, we spend one month running different scenarios with the radio controlled cars and changing experimental parameters such as the epoch time per trial circle helping us to understand better the nature of the wireless sensor network we deployed and evaluate better the engine of the In-Motes Reloaded middleware. Packet delivery did not affect by the distance between the aggregation point and the sensors rather from facts such as the duration per measuring period and buffer overflows at the facilitator node end.

Figure 6-15 presents the total number of packets that were delivered from a facilitator node monitoring one car that breached the acceleration critical parameter for different epochs.

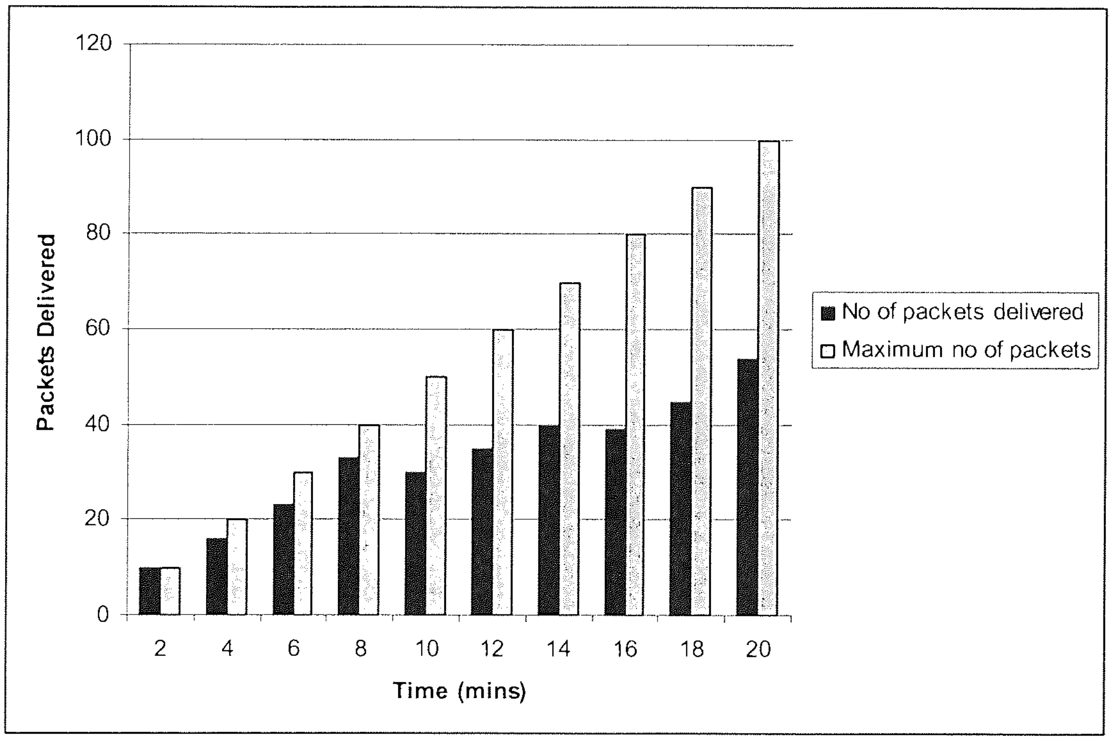


Figure 6-15: Packet delivery performance of the middleware running the In-Motes EYE for different measuring periods

From the above graph it is obvious that the packet delivery performance was affected as the measuring period was increased. Packet losses were observed mainly due to two reasons. Firstly, increasing the activity of a slave node of a radio controlled car it meant that we were increasing the battery power consumption of the mote. For mica2 motes this increase usually affects the performance of the hardware resulting to delays in obtaining a sensing value and even stalls the operation of a node as the battery levels are exhausting. Secondly, the facilitator node when the traffic is heavy it will drop some packets as its sending buffer will overflow.

Although, those problems affected the performance of the middleware we observed a success rate that was above 50% in all the trials. That can be explained due to the modifications that we applied to the core engine of the middleware eliminating in most cases race conditions, re-transmissions and overuse of memory resources both virtual and physical ones.

Overall, we observed a better behavior of the In-Motes Reloaded middleware than this of In-Motes for the duration of the trial as can be seen bellow:

8. Stall of the whole network: 5
9. AgentSender Fail sending agent: 12
10. Null Readings: 100
11. Facilitator Buffer_Overflow: 5
12. Node Buffer_Overflow: 6
13. AgentReceiver Fail receiving agent: 15
14. Misplacement of sensor/Drop/Other: 20

Failures that as before solved instantly by our middleware the moment they were noticed, by simply flashing the sensors and reinstalling remotely the application. Stalls of the 2 sensors of the radio controlled cars we were using were not observed frequently mainly cause of the middleware ability to handle and allocate better the memory resources of the system than before. The facilitator scheme and In-Motes Reloaded communication protocol worked as expected and we did not observe any store and forward delays.

6.7 Summary

This chapter presents the updated version of In-Motes middleware, which we named In-Motes Reloaded, core programming changes, testing and evaluation. The In-Motes Bins trial and the lessons that we learned by running the application in a real environment helped us to update some of the core functionalities of the middlewares engine in an effort to resolve issues such as poor packet delivery performance and internal race conditions. Initially we carry out our experiments in the controlled environment of the laboratory where we observed the behaviour of the wireless sensor network under the new middleware engine. We compared and analysed our approach using the SenseLight application, in order to see where our new middleware stands in comparison with the older approach. Moving one step further we created a complex application, the In-Motes EYE, that was deployed in an open environment and was measuring acceleration variation of moving objects, in our case radio controlled cars. Our successful implementations of the In-Motes EYE application together with the steady performance of the new version of the middleware compared with the previous version, lead us to envisage a near future where the wireless sensor technology could establish a framework that will overcome various limitations that those networks inhabit.

CHAPTER 7

Conclusions

The findings of the thesis are summarized in this chapter. They consist of evaluation, conclusions and observations, followed by suggestions for future research.

7.1 Aims of the thesis

The aims of the thesis were:

1. To investigate how we can increase the reliability, fault tolerance and flexibility of a wireless sensor network by focusing mainly on the application layer of these networks.
   (a) Investigate the wireless sensor network theory and identify key limitations of this technology.
   (b) Identify architectures and approaches in wireless sensor networks by categorizing them in terms of functionality and organization
   (c) Focus on the application layer and investigate the middleware approach as a viable solution for overcoming the limitations that wireless sensor network inhabit.
2. To develop a method that will be able to moderate effectively the interconnections between the wireless nodes and that will provide recovery mechanisms incase of a failure.

(a) Devise a dynamic communication protocol that will be able to incorporate intelligence in terms of perception cognition and control to every active node in the wireless sensor network

I. Incorporate mobile code/agents.

II. Incorporate Linda-Like tuple spaces.

III. Add behavioural rules based on a model of bacterial strains.

3. To develop energy efficient method that will be able to provide remote mechanisms for controlling and monitoring the wireless sensor network during the lifetime of an application.

(a) Develop a prototype platform(s) for deploying applications in the wireless sensor network.

(b) Incorporate the communication protocol in the system(s)

(c) Develop a comparison model for evaluating this approach against a number of other approaches that exist in the same category

(d) To develop a number of prototype applications for experiments and testing in order the end system(s) to be evaluated.

7.2 Evaluation

7.2.1 Wirelesses sensor network issues

The literature in the field of wireless sensor networks revealed the fundamentals of sensor networks organization and functionality. Constraints such as energy, flexibility in re-programming and fault tolerance were made apparent as well as the need to effectively bridge the gap between the applications and low level constructs such as the physical layer of those networks. The middleware approach seems as a promising way of achieving the above but again the systems we reviewed raised a number of consideration that need to be addressed before we can conclude that this approach is a viable one.

Firstly, most of the middlewares were based upon a virtual grid format that the developers will try to imitate when they will deploy an application in a real world. Thus, the whole system will be quite inflexible and susceptible to errors since real world environments tend to be quite dynamic and the locality of the nodes can be altered due to unexpected scenarios very easily. Secondly, the approaches we reviewed did not incorporate any fault tolerant mechanisms incase of a node failure. Thus, the smallest malfunction of a node can lead to catastrophic scenarios where a wireless sensor network will stall its operation. Last, but not least, the communication protocols that they adapted proved to be quite energy consuming and dysfunctional since in most cases a flooding of network was a necessity for altering the simplest parameter of the network and based on loose hierarchy internal race conditions and buffer overflows were taking place leading to poor package delivery and eventually the stall of individual nodes. All those considerations were taken into account when we started the development of our approach. This completed the Aim 1 of the thesis.

7.2.2 The key terminologies of our approach

The literature in areas such as mobility of a code and Linda-like tuple spaces was investigated in an effort to focus on ways in creating a dynamic and flexible communication protocol for the wireless sensor nodes. Using mobile code in the form of our In-Motes agents, we were able to transfer small fragments of code to every node in the wireless sensor network thus making the post development reprogramming feasible. The mobile code inside our network had a specific character defined by their development specification, a character that was accumulated from every node they were occupying. In-Motes agent communication was achieved by adapting a facilitator model which states that a predetermined number of agents communicate with a specific facilitator agent thus forming a number of small "federations" inside our network. Each "federation" was able to communicate with another existing one only through their facilitators (Aim 2(a) I). In order to increase the flexibility of the communication protocol a list of behavioural rules based on bacterial strains was compiled and allocated to each facilitator In-Motes agent. This added hierarchy was useful for overcoming problems such as buffer overflows and packet delays (Aim 2(a) III).

The next step was to create a communication link between the mobile code and the actual hardware. This was achieved by making use of the Linda-like tuple space scheme. This virtual shared memory framework was accessible to all the running processes of the network. Added reactions in the tuple space were generated so that each tuple resident in a node could have its own state and react to specific actions provided by the hosted In-Motes agent. Reactions by having permission to access the tuple space could change the context and modify the semantics of the visited In-Motes agents thus altering a number of parameters such as bandwidth and packet frequency for every node (Aim 2(a) II). Overall, the scheme was providing better control between the interactions of the hardware and the mobile code.

7.2.3 The proposed middleware for wireless sensor networks

In Chapter 4, our proposed middleware, In-Motes, was outlined and described, exhibiting how technologies such as mobility of a code and Linda-like tuple spaces could be combined and integrated under the same framework. The main idea was to create a flexible and energy efficient platform for deploying applications in a wireless sensor network (Aim 3(a)). The communication protocol, which our theoretical analysis in the previous chapter verified its usefulness in terms of monitor and control of the wireless sensor nodes, was successfully incorporated in the middleware's engine (Aim 3(b)).

A number of preliminary tests were performed in the laboratory control environment in order to identify if any problems with the programming code of the middleware existed. Small trials aiming to verify the correct delivery of the mobile code and their ability for single and multi hopping migration were successfully implemented. Since no major faults were reported we moved in the next step which was the creation of a simple application, SenseLight, which was deployed in a small scale network through our middleware. SenseLight is a multi-hop application that uses mobile code to collect light readings and report back to the end user. The application and small variations of it was deployed in the wireless sensor network by using the In-Motes middleware, the Agilla middleware and the TinyOS framework respectively, thus meeting Aim 3(c). This comparison model identified that our approach was more efficient in terms of packet delivery than the other two but also that both In-Motes and Agilla middlewares were experiencing problems such as stall of sensor nodes, after a period above 2 minutes of continuously sending data to the aggregation point.

In Chapter 5, the In-Motes Bins application was presented as the first complex application of our research that was deployed in a large scale wireless sensor network in a dynamic everyday real time environment. In-Motes Bins is a mobile code based real time application developed for monitoring environmental conditions and more specifically variations in light and temperature and it was implemented for 4 months in a wine store Aim 3(d). The trial was divided in two parts and a number of schemes concerned with the typology of the mobile code and migration techniques were applied in an effort to increase the efficiency of our middleware at an application level. The advantages of what we named dynamic In-Motes job agents against static In-Motes job agents were identified as well as the limitations that cloning transmission of the mobile code inhabits when compared with multi-hop migration in terms of conserving energy. The merits of the communication protocol were tested as well and no problems were identified to that extend. The dynamic In-Motes job agents increased the packet delivery performance and the efficiency in energy resources of the network although our approach suffered from packet losses and stalls of nodes mainly due to internal race conditions and buffer overflows.

Moreover, the wireless sensor network was active for a longer period than in the laboratory trials, so we had a clear picture of its behavior. We concluded at that stage that both the battery levels of the individual motes and the In-Motes tuplespace communication could cause in time unexpected errors that we listed in Chapter 5. Overall, the In-Motes Bins application successfully tested the merits of our middleware in a very dynamic environment and even provides a practical solution for a problem that the store was facing prior to our visit.

The lessons learned from the In-Motes Bins trial were very useful for the further development of our middleware. We applied a number of modifications in some of the programming parameters of the In-Motes engine such as the number of active mobile per node and the size of tuples while in parallel programming code blocks were added to both nesC and Java packages aiming to improve the overall efficiency in terms of energy usage and packet delivery of our approach. Thus, In-Motes Reloaded was developed as an updated version of In-Motes. In order to test the merits of the new middleware we developed the In-Motes EYE application. In-Motes EYE is a mobile code based real time application developed for monitoring variations in acceleration of moving objects Aim 3(d). Compared to the In-Motes middleware the new version proved to be superior, since the packet delivery performance was dramatically increased and most of the buffer overflows and internal race conditions that were reported before were mostly eliminated. The recovery mechanisms incase of failures and unexpected scenarios were as before very steady and efficient since the whole network and the deployed application could be rebooted in less than 1 minute.

7.3 Recommendations for future research

7.3.1 Prototype middleware approach

Perhaps one of the most important issues that would render the proposed system truly applicable for real time applications is the automatic process and analysis of the obtained data deriving from the wireless nodes. During our trial with both In-Motes Bins and In-Motes EYE application, the sensing data that was delivered to the base station was processed and categorized into useful information by transforming it to meaningful engineering units manually, since it would not have any impact to the experimental results. Future research would investigate whether this process could be automated. This involves additional programming and raises questions such as where this process would take place and what impact would have to the energy levels of the nodes in the scenario where it would be placed at the facilitator level.

When we conducted trials in the laboratory controlled environment we noticed poor performance of the systems in terms of packet loss and unexpected stalls of the nodes of the wireless sensor network, something that surprisingly enough was not observed in real time dynamic environments to that extend. It was then conjectured that one of the possible reasons for this phenomenon was interference deriving from the other electronic units that were active in the laboratory. Other possible reasons were the battery levels of the individual motes and the tuplespace communication scheme that we adapted in our platform. An investigation aiming to identify and prove the reasons for this interference would considerably help and perhaps aid the inclusion of a correct radio transmission set up for the nodes. Also further testing could be applied for a small scale wireless network for a long continuous period in order to record better the behavior of the nodes in terms of performance.

In both In-Motes Bins and In-Motes EYE we used the same communication protocol. The facilitator scheme was used for the inner communication, the Linda-like tuple spaces for the mobile code-node interactions and finally the list of the behavioural rules for added hierarchy of the facilitator nodes all in an effort to make a flexible platform for deploying applications, This choice was justify in our trials by a good balancing between packet/recovery transmissions and complexity. However, other communication protocols could be implemented for the middleware such as the GAF communication scheme. Future research could experiment with more protocols, in order to establish the best one available.

The energy levels of the individual motes were not recorded rather than observed concluding that as the time increases and the battery levels drop, motes can stall their operation with higher frequency. Thus, a more substantial test is needed in order the above trial observation/assumption to be verified.

Last, but not least, both systems were developed and applied without any security framework build in for data protection. Wireless sensor network nodes had a specific id and were transmitting data over a predefined user selected radio; however this is not secure and could be easily tampered from an intruder. Future research could incorporate security mechanisms for protecting the application data from a number of attacks.

7.4 Conclusions

Overall, the research has achieved its aims, providing two middleware solutions for flexible deploying applications in a wireless sensor network with both of them incorporating an energy efficient and adaptable communication protocol. Both systems showed good performance for trials that were executed in real dynamic everyday environments.

Regarding, the two prototype applications that were developed as part of the middleware evaluation, their success reflected upon the fact that with the right infrastructure a wireless sensor network implementation can be incorporated in our everyday lives and provide us with adequate solutions for complex problems.

It was shown through our research that the middleware approach is a promising way of overcoming the limitations that the wireless sensor networks inhabit. Our middleware approaches addressed and resolved some of those limitations in a satisfying way.

Last, but not least, there is a rule that applies to every wireless sensor network approach. That is, regardless of the amount of formal proofs and computer simulations which support its applicability, it should be tested under real dynamic environments in an extensive way. Only in this way can any unforeseen flaws be detected, which will lead to either the creation of a revised version of the scheme, or its rejection. The developed communication protocols are no exceptions.

Bibliography

Agent Tcl, Transportable Agent System. Computer Science Sept., Dartmouth University 1995, available at: http://www.cs.dartmouth.edu/agenttcl.html [2/08/2007]

Agre, P & Chapman, D (1987). "PENGI: An Implementation of a Theory of Activity", *In Proceedings of the 6th National Conference on Artificial Intelligence,* AAAI-87, Seattle, WA. pp. 268-272.

Ahmad, Khurshid (1995). "A Knowledge-based Approach to the Safe Design of Distributed Networks", *Proceedings of the Safety-critical Systems Symposium*, Brighton. London: Springer-Verlag Ltd. pp. 290-301.

Aparicio, G (1996). "IBM Intelligent Agents", *FIPA Opening Forum Proceedings*, Yorktown, New York, 1996.

Akyildiz I.F, Y. Sankarasubramaniam W. Su, Cayirci E (2002), 'Wireless sensor networks: a survey', *Computer Networks*, pp. 393–422.

Analog Devices (2007), "ADXL202 Low-Cost ±2 g Dual-Axis Accelerometer with Duty Cycle Output", web site, http://www.analog.com, [13/03/2007]

Barbuceanu, Mihai (1997). "Co-ordinating agents by role based social constraints and conversation plans". *AAAI-97 (Fourteenth National Conference on Artificial Intelligence)*, IAAI-97, Providence, Rhode Island. pp. 62-69.

Bond, Allen, H & Gasser, Les (1988). "An Analysis of Problems and Research in DAI". *In (Eds.) Readings in Distributed Artificial Intelligence.* Los Angles: Morgan Kaufmann. pp 3-35

BIBLIOGRAPHY

Bradshaw, M, Jeffrey (1997). "An introduction in software agents", *In (Eds.) Bradshaw, M, Jeffrey in Software Agents*, The MIT Press.

Berkeley (2001), University of California, "800 nodes self-organized wireless sensor Network", http://today.cs.berkeley.edu/800demo/ [13/03/2007]

Carriero Nicholas, David Gelernter (1989), "Linda in Context", *Communications of the ACM,* pp. 444 – 458.

Coen, Nichael, D (1994). "Sodabot: A Software Agent Environment and Construction System", *Proceedings of the CIKM Workshop on Intelligent Information Agents*, available at http://citeseer.ist.psu.edu/coen94sodabot.html [2/08/2006]

Chaib-draa, B., & Moulin, P (1987). "Architecture for Distributed Artificial Intelligent Systems," *In IEEE Proceedings*, Montreal, pp. 64-69.

Cabri Giacomo, Leonardi Letizia, Zambonelli Franco (1998), "Reactive Tuple Spaces for Mobile Agent Coordination", *Proceedings of the Second International Workshop on Mobile Agents,* pp. 237 - 248

Culler D., Estrin D., Srivastava M. (2004), "Overview of Sensor Networks" *IEEE Computer*, pp. 41-49.

Crossbow (2007), "Wireless Systems for Environmental Monitoring", web site, http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Smart_Dust_AppNote.pdf, [13/03/2007]

Dunkels Adam, Schmidt Oliver, Voigt Thiemo, Ali. Muneeb (2006), "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems", *Proceedings of the 4th international conference on Embedded networked sensor systems,* pp. 29-42.

BIBLIOGRAPHY

Dunkels Adam, Finne Niclas, Eriksson Joakim, Voigt Thiemo (2006), "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks", *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems,* pp. 15-28.

Esquirol, Patrick, Fargier, Helene, Lopez, Pierre & Schieux,(1996). "Constraint Programming". In Belgian Journal of Operational Research, Statistics, and Computer Science, 1996, available at http://citeseer.ist.psu.edu/esquirol95constraint.html [2/08/2004]

Fok Chien-Liang, Roman Lu Gruia (2005), "Rapid Development and Flexible Deployment of Adaptive Network Applications", *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems,* pp. 653-652.

Fok Chien-Liang, Roman Lu Gruia (2005), "Mobile Agent Middleware for Sensor Networks: An Application Case Study", *Proceedings of the 4th International Conference on Information Processing in Sensor Networks,* pp. 382-387.

Franklin, Stan & Graesser, Art (1996). "Is it an agent, or just a program? A Taxonomy for autonomous agents", In *Proceedings of the Third International Workshop on Agent theories, Architectures, and Languages,* Springer-verlag. pp 21-35.

Ferguson, I, A (1992). "TouringMachines : An Architecture for Dynamic, Rational, Mobile Agents", *Ph.D. Thesis*, Clare Hall, University of Cambridge, UK.

Gay David, Levis Philip, Behren Robert (2003), "The nesC Language: A Holistic Approach to Networked Embedded Systems", *ACM SIGPLAN Conference on Programming Language Design and Implementation,* pp. 400-417.

BIBLIOGRAPHY

Georgoulas Dimitrios, Blow Keith (2006), "In-Motes: An Intelligent Agent Based Middleware for Wireless Sensor Networks", Best Student Paper Award, *Proceedings of the 5th WSEAS International Conference SEPADS 06*, pp. 225-231.

Georgoulas Dimitrios, Blow Keith (2006), "Making Motes Intelligent: An agent based approach to wireless sensor networks", *In WSEAS on Communications Journal,* pp. 515-522.

Georgoulas Dimitrios, Blow Keith (2007), "In-Motes Bins: A Real Time Application for Environmental Monitoring in Wireless Sensor Networks", *Proceedings of the 9th IEEE/IFIP International Conference on Mobile and Wireless Communications Networks*, pp21-26.

Gehrke Johannes, Madden Samuel (2004), "Query processing in Sensor networks", *Pervasive Computing,* Published by the IEEE CS and IEEE ComSoc, pp. 200-210.

Gummadi Ramakrishna, Gnawali Omprakash, Govindan Ramesh (2005), "Macro programming Wireless Sensor Networks using Kairos", *Proceedings of the International Conference on Distributed Computing in Sensor Systems,* pp. 130-145.

Genesereth, M, R & Ketckpel, S, P. (1994), "Software Agents", *In Communications of the ACM 37(7)*, pp. 48-53.

Hart Jane K, Martinez Kirk (2005), "Environmental Sensor Networks: A revolution in the earth system science?", *Earth-Science Reviews 78,* pp. 177–191.

Heinzelman Wendi, Chandrakasan Anantha, Balakrishan Hari (2000), "Energy Efficient Communication Protocol for Wireless Microsensors Networks", *Proceedings of the Hawaii International Conference on System Science,* pp. 10-17.

BIBLIOGRAPHY

Hill J., Szewczyk R., Woo A., Hollar S., D. Culler (2000), " System Architecture directions for Networked Sensors", *Architectural Support for Programming Languages and Operating Systems*, pp. 93-104.

Intel Corporation (2006), "Wireless Vineyard Research Project", web site, http://www.intel.com/research/vert_agri_vineyard.htm [13/03/2007]

Johnson Derek M, Teredesai Ankur M, and Saltarelli Robert T (2005), "Genetic Programming in Wireless Sensor Networks", *Proceedings of the 8th European Conference on Genetic Programming*, pp. 96-107.

Jolly Vasu, Latifi Shahram (2006), "Comprehensive Study of Routing Management in Wireless Sensor Networks- Part-2", *Proceedings of the 2006 World Congress in Computer Science Computer Engineering, and Applied Computing*, pp. 400-411.

Kim Tae-Hyung and Hong Seongsoo (2003), "Sensor Network Management Protocol for State-driven Execution Environment", *Proceedings of the International Conference on Ubiquitous Computing*, pp. 197-199.

Levis Philip (2004), "Viral Code Propagation in Wireless Sensor Networks", *UC Berkeley Tech Report*, UCB//CSD-04-1350.

Levis Philip and Culler David (2002), 'Mate: A Tiny Virtual Machine for Sensor Networks', *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, pp. 100-111.

Lee Winnie Louis, Datta Amitava, Cardell-Oliver Rachel (2006), 'WinMS: Wireless Sensor Network-Management System' *CSSE Technical Report*, UWA-CSSE-06-001

BIBLIOGRAPHY

Lee Winnie Louis, Datta Amitava, Cardell-Oliver Rachel (2007), 'Network Management in Wireless Sensor Networks', *Handbook of Mobile Ad Hoc and Pervasive Communications,* accepted to appear.

Levis Philip (2006), "TinyOS programming", manual, Stanford University, http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf [13/03/2007]

Marsh David, Ruzzelli Antonio, Tynan Richard (2005), 'Agents for Wireless Sensor Network Power Management', *Proceedings of the Parallel Processing ICPP,* pp. 413-418.

Martinez K, Padhy P, Elsaify A, Zou G, Hart J.K (2006), 'Deploying a Sensor Network in an Extreme Environment', *Proceedings of the Sensor Networks, Ubiquitous and Trustworthy Computing International Conference Hawaii,* pp. 186-193.

Madden Sam, Hellerstein Joe, Hong Wei (2003), 'TinyDB: In-Network Query Processing in TinyOS', *ACM Transactions on Database Systems,* pp. 122-173.

Nwana, S, Hyacinth (1996). "Software Agents: An Overview", *In Knowledge Engineering Review,* Vol. 11, No. 3, pp. 205-244.

Omicini Andrea, Zambonelli Franco (1998), "TuCSoN: a Coordination Model for Mobile Information Agents", *Proceedings of the 1st Workshop on Innovative Internet Information Systems,* pp. 123 - 132.

Panzarasa, P., Jennings, N. R. and Norman, T. J. (2002) "Formalising Collaborative Decision Making and Practical Reasoning in Multi-Agent Systems", pp. 55-117

BIBLIOGRAPHY

Ramanathan Nithya, Chang Kevin, Kapur Rahul, Girod Lewis, Kohler Eddie (2005), "Sympathy for the Sensor Network Debugger", *Centre for Embedded Network Sensing,* Paper 98.

Ruiz Linnyer Beatrys, Nogueira Jose Marcos (2003), "MANNA: Management Architecture for Wireless Sensor Networks", *Communications Magazine,* Volume 41, Issue 2, pp. 116 – 125.

Romer Kay, Kasten Oliver, Mattern Friedemann (), "Middleware Challenges for Wireless Sensor Networks", *CM SIGMOBILE Mobile Computing and Communications Review,* pp. 59-61.

RoadKnight, Marshall Ian (2000), "Future Network Management – A bacterium inspired solution", *Proccedings of the 2nd International Symposium Engineering and Intelligent Systems*, pp. 123-129

Salem Hadim, Nader Mohamed (2006), "Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks", *IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922,* Vol. 7, No. 3.

Song Hyungjoo, Kim Daeyoung, Kangwoo Lee, Jongwoo Sung (2005), "UPnP-Based Sensor Network Management Architecture", *Proceedings of the second International Conference on Mobile Computing and Ubiquitous Networking,* pp. 85-92.

Song Dezhen (2005), "Probabilistic Modelling of Leach Protocol and Computing Sensor Energy Consumption Rate in Sensor Networks", *Lecture Notes in Computer Science,* Volume 3121/2005, pp. 157-170.

BIBLIOGRAPHY

Souto Eduardo, Guimarães Germano, Vasconcelos Glauco (2004), "A Message-Oriented Middleware for Sensor Networks", *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing,* pp. 127-134.

Sycara, K, P (1989). "Multi-agent Comprise via Negotiation". *In (Eds) Gasser and Huhns. Distributed Artificial intelligence,* pp. 119-137.

Tegmo-Larsson and Spittler (1990), "Temperature and Light Effects on Ethyl Carbonate Formation in Wine during Storage", *In Journal of Food and Science Volume 55*, www.blackwell-synergy.com/doi/abs/10.1111/, [13/03/2007]

Wavish, P. & Graham, M. (1994), "Roles, Skills and Behaviour", *In (Eds.) Wooldridge, M. & Jennings*, N. (1995b), Intelligent Agents, Lecture Notes in Artificial Intelligence 890, Heidelberg: Springer Verlag, pp. 371-386

Wooldridge Michael, Jennings Nicholas, R (1995). "Agent Theories, Architectures, and Languages: a Survey", *In (Eds.) Wooldridge and Jennings, Intelligent Agents*, Berlin: Springer-Verlag, pp. 1-22.

Wilkinson Andrew (2005), "About Linda", *Online Documentation*, http://www-users.cs.york.ac.uk/~aw/pylinda/about.html [13/03/2007]

Zhou, Lingzhong, Thornton, John & Sattar, Abdul (2003). "Dynamic Agent Ordering in Distributed Constraint Satisfaction Problems", *Proceedings of Australian Conference on Artificial Intelligence 2003*, Springer 2003, available at http://miami.int.gu.edu.au/dbs/3016/john/publications/AI2003lingzhong.pdf [13/03/2007]

Zhao J., Culler D. (2003), "Understanding packet delivery performance in dense wireless sensor networks", *Proceedings of the ACM SenSym*, pp. 23-31

Appendices

Appendix A

In-Motes Instruction Set

(Includes both In-Motes and In-Motes Reloaded Instruction Set Architecture)

Instruction Set Architecture

Instruction Classes

Class Format Key
b-class 00ii iiii i = instruction
t-class 0100 ixxx i = instruction, x = argument
e-class* 0101 iiii xxxx xxxx xxxx xxxx i = instruction, x = argument
v-class 011i xxxx i = instruction, x = argument
x-class 11xx xxxx x = argument
* stored in big-endian format

Instruction Categories

- General purpose instruction
- Facilitator Instruction
- Slave Instruction
- Job Instruction
- Fix Instruction
- Sets condition code
- Accessor to acquaintance list
- Migration instruction
- Accessor to the heap
- Accessor to sensors
- Local tuple space operation
- Remote tuple space operation

B-class Instructions [00ii iiii]

- **halt** Halt execution 0x00 Halts execution of the issuing agent. This frees the agent's context for another incomming agent. Used for all the In-Motes agents.

- **jobhalt** Halt execution 0x01 Halts execution of the job agent. This frees the agent's context for another incomming job agent.

- **loc** Push host location 0x02 [location] Pushes the mote's location onto the operand stack as a value. Used for all the In-Motes agents.

- **aid** Push agent ID 0x03 [agentid] Pushes agent's ID onto the operand stack.

- **rand** Push random number 0x04 [value] Pushes a 16-bit random number onto the operand stack as a value.

- **cpush** Pushes condition onto the stack 0x05 [value] Pushes the issuing agent's condition variable onto its stack as a value. Used for all the In-Motes agents.

- **depth** Push stack depth 0x06 [value] Pushes the depth of the agent's operand stack onto the stack as a value; the depth is the value before executing this operation.

- **vicinity** Checks if dist <= 2 0x07 [location] Checks whether the location on the stack is within a distance of <= 2 of the location(s) on the heap. Heap [0] must be the number of locations, Heap[1...n] must be the locations.
- **clear** Clear the operand stack 0x08 Clear the operand stack.
- **numslave** number of slaves 0x09 [value] Pushes the number of slaves onto the stack.
- **wait** make agent wait 0x0a Stops agent execution without deallocating its resources. Allows it to wait for a reaction to fire
- **fwait** make a facilitator agent wait 0x0a Stops agent execution without deallocating its resources. Allows it to wait for a reaction to fire
- **inc** Increment value 0x0b [value] [value+1] Pops a value off the operand stack and pushes the incremented value back onto the stack.
- **clearvar** Clear heap variable 0x0c [value] Pops a heap address value off the operand stack and clears the heap at that address.
- **inv** Invert a value 0x0d [value] [value] Pops a value off the operand stack, multiplies it by -1, and pushes the result onto the stack.
- **not Boolean** not 0x0e [value] [value] Pops a value off the operand stack. If the value is not 0, push a 0 onto the stack, else push a 1.
- **lnot** logical not 0x0f [value] [value] Pops a value off the operand stack and pushes its logical not (e.g. 0x2e will result in 0xffd1).
- **copy** Copy top of operand stack 0x10 [var] [var], [var] Pops a variable off the operand stack, pushes it back onto the stack twice.
- **pop** Pop top of operand stack 0x11 [var] Pops a variable off the operand stack.
- **cpull** Pulls value from stack onto condition 0x12 [value] Pops the value off the stack and sets it to be the condition.
- **sleep** Sleep 0x13 [value] Halt the slave agent for [value] * 0.125 seconds.
- **fsleep** Sleep 0x14 [value] Halt the facilitator agent for [value] * 0.125 seconds.
- **putled** Actuate LEDs 0x15 [value] or [reading] Takes a single operand and lights the LEDs as follows. If the operand is a value: It uses the lowest five bits of the operand to determine how to actuate the LEDs. The lowest three bits denote the 3 LEDs; bit 0 is red, bit 1 is green, and bit 2 is yellow. The next two bits (3 and 4) specify which operation to apply; 00 is set, 01 is off (active on 0), 10 is on, and 11 is toggle.

Operation	Binary Value	Decimal Value
set only red on	00001	1
set only green on	00010	2
set only yellow on	00100	4
Toggle red	11001	25
Toggle green	11010	26
Toggle yellow	11100	28
Toggle all 3 LEDs	11111	31
Turn off red	01110	14
Turn off green	01101	13
Turn off yellow	01011	11
Turn off all 3 LEDs	01000	8
Turn on red	10001	17
Turn on green	10010	18
Turn on yellow	10100	20
Turn on all LEDs	10111	23

If the operand is a reading, it simply displays the lower 3 bits in on the LEDs.

- **smove** Slave migration 0x16 [location] Performs a slave migration to a remote host. The location of the remote host is specified by the parameter on top of the operand stack. A slave migration does not affect the execution of the agent (the pc, stack, and heap are all maintained). The condition code is set as follows: 0 - move failed (agent continues to run on original host/slave) 1 - move succeeded (agent now running on remote host)

- **fmove** Facilitator migration 0x17 [location] A facilitator version of smove.

- **sclone** Slave clone 0x18 [location] Pops the value off the operand stack. Clones itself at node with address value. The clone inherits the initiating agent's program counter. The condition code is set as follows: 0 - clone failed (agent continues to run on original host) 1 - clone succeeded (agent now running on remote host) 2 - child (this is the code set on the child agent prior to running)

- **fclone** Weak clone 0x1a [location] A facilitator version of sclone.

- **getvars** get a heap var, address in stack 0x1b [value] [var] Retrieves a variable off the heap and pushes it onto the operand stack. The address of the heap is specified by the value on top of the stack.

- **setvars** set heap var, address in stack 0x1c [value] [var] Saves a variable on the heap. The variable and address to store it must both be on the operand stack.

- **getslave** get federations slave address 0x1d [value] [location] Get the address of the slave neighbor. All neighbors are stored in a list. The value specifies which position in the list to get. The value must be between 0 and numnbrs- 1. Sets condition = 1 if success, else sets condition = 0.

- **sense** take a sensor reading 0x1e [value] [reading] Reads a sensor. The sensor type is specified by the value on top of the operand stack, defined as follows: Photo = 1 Temp = 2 Microphone = 3 Magnometer X = 4 Magnometer Y= 5 Accelerometer X =6 Accelerometer Y= 7 Sounder = 8

- **dist** distance 0x19 [reading1], [reading2] [value]Pops two variables off the operand stack. The two variables are both sensor readings, this calculates how far apart they are.

- **swap** Swap top two variables 0x20 [var1],[var2][var2],[var1]Pop [var1] then [var2] off the stack, then push [var2] then [var1] back on stack. This swaps their positions on the stack.
- **mul** Multiply 0x21 [variable1],[variable2]][variable1]*[variable2] Pops two variables off the stack and performs a multiplication. Valid parameters are:
- **div** Divide 0x22 [variable1], [variable2] [variable2] ----------- [variable1] Pops two variables off the stack and performs a division operation. Valid parameters are: Two values A value (variable 1) and a sensor reading (variable2)
- **add** Add two values 0x23 [value1], [value2] [value] Pop [value1] and [value2] off the stack, push [value1] + [value2] onto the stack.
- **cgt** Greater than 0x2e [Value1],[Value2] or[Reading1],[Reading2] Pops the top two values off the stack. Set the condition to true if [Value2] > [Value1] or [Reading2] >[Reading1], false otherwise
- **out** out on host tuple space 0x24 [tuple] Pops a tuple off the operand stack and places it into the local tuple space. This tuple become a public tuple accessible to anybody.
- **inp** inp on host tuple space 0x25 [template] [tuple] ? Searches the host's local tuple space for a tuple matching a template. If a match is found, remove it from the tuple space, push its fields onto the operand stack and set the condition to 1. If no matching tuple is found, set condition to 0.
- **rdp** rdp on host tuple space 0x26 [template] [tuple] ? Same as inp except do not remove tuple.
- **in** in on host tuple space 0x27 [template] [tuple] Same as inp except blocks until a tuple is found.
- **rd** rd on host tuple space 0x28 [template] [tuple] Same as rdp except blocks until a tuple is found.
- **srd** rd on slave tuple space 0x28 [template] [tuple] Same as rdp except blocks until a tuple is found for slave agent.
- **tcount** counts the number of tuples in the local tuple space that match a template x29 [template] [value] Pops a template off the op stack and pushes the number f tuples that match it onto the op stack. This operation only considers tuples that are public, system, or private o the executing agent.
- **rout** remote OUT 0x30 [location], [tuple] Inserts a tuple into a remote hosts's tuple space. Sets cond=1 if successful, 0 otherwise. The tuple becomes a public tuple accessible to anybody.
- **fout** remote OUT 0x31 [location], [tuple] Inserts a tuple into a facilitators tuple space. Sets cond=1 if successful, 0 otherwise. The tuple becomes a public tuple accessible to anybody.
- **regrxn** register reaction 0x3e [value], [template] Registers a reaction on the tuplespace. The template specifies the type of tuple that causes the reaction to fire. The value specifies what the program counter should be set to when the reaction fires. Reactions are carried across strong migrations and clones.

- **deregrxn** deregister reaction 0x3f [template] Deregisters a reaction from the tuplespace.

- **getslv** 0x31 [value] [id] gets a slave id.

T-Class Instructions [0100 ixxx]

- **pushrt** push reading type 0x40-0x47 [type] Pushes a reading type onto the operand stack.001 - photo reading [PHOTO]010 - temperature reading [TEMP] 011 - microphone reading [MIC]100 - magnometer x-axis reading [MAGX]101 - magnometer y-axis reading [MAGY] 110 - accelerometer x-axis reading [ACCELX]111 - accelerometer y-axis reading [ACCELY]

- **pusht** Push type 0x48-0x4d [type] Pushes a type onto the operand stack.000 – ANY 001 – AGENTID 010 – STRING 011 – TYPE 100 – VALUE

E-Class Instructions [0101 iiii xxxx xxxx xxxx xxxx]

- **pushn** Push a name onto the operand stack 0x31 [name] Pushes a name onto the operand stack. A name is a 16-bit value with the following format: [a-z,0-9][az,0-9][a-z,0-9]. It is encoded as follows: [a = 1, b = 2, ...,0 = 26, 1 = 27, ...].

- **pushcl** Push constant long 0x32 [value] Pushes a 16-bit value onto the operand stack. This is more powerful than pushc since pushc can only push a 6-bit value onto the operand stack.

V-Class Instructions [011i xxxx]

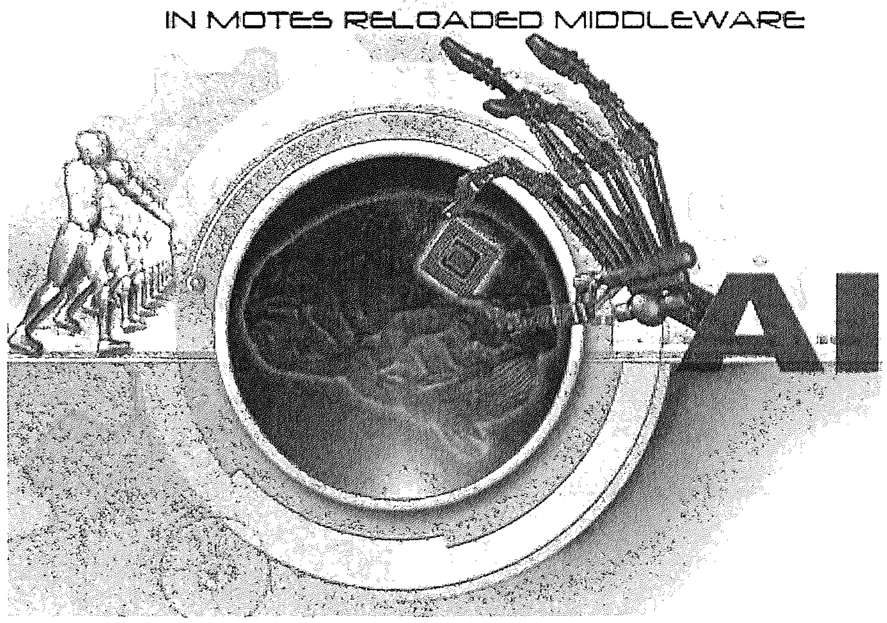
- **getvar** moves a variable from the heap to the stack 0x60 - 0x6b [var] Copies a variable from the heap and pushes it onto the stack. The address within the heap is specified by the last 4 bits of the instruction. Since there is only a 12-word heap, only addresses 0-b are valid. If the heap variable is INVALID, it does not push anything onto the stack and sets condition=0. Otherwise, it sets condition=1. 0x6c - 0x6f

- **setvar** Moves a variable from the stack to the heap 0x70 - 0x7b [var] Pops a variable off the operand stack and puts it into the heap. The address within the heap is specified by the last 4 bits of the instruction. Since there is only a 12-word heap, only addresses 0-b are valid. 0x7c-0x7f

X-Class Instructions [11xx xxxx]

- **pushc** Push a constant value onto the stack 0xc0 - 0xff [value] Pushes the least signicant 6 bits onto the operand stack as a value (range is 0 to 63).

In-Motes Reloaded User Guide



✦ Installing In-Motes Reloaded middleware in your computer

The following steps will install the In-Motes Reloaded middleware in your computer. The middleware is applicable for all the versions of the TinyOS operating system and was tested using Windows XP and earlier versions. A user should not have any problems with Linux systems but in the case of Windows Vista make sure that you have set yourself as the administrator of the machine you are using otherwise access problems will occur.

1. Install TinyOS operating system in your computer
2. In-Motes Reloaded comes as two separate packages with a portion of code in NesC and a second one in Java. Unzip both of them in an area of your computer. They exist in the In-Motes Reloaded folder of the accompanied CD.
3. Create a folder myApps in the TinyOS root path /opt/tinyos-1.x
4. Copy/Paste the portion of the NesC code and add it to the myApps folder
5. Copy/Paste the portion of the Java code and add it to the root path /tools/java
6. Download the makelocal file (displayed below) and install it in /tools/make. Customize the radio frequency, group address and serial port number defined within it. The group address should be unique to you. The serial port number is the port that your mote programming board is connected to.

```
# Radio Frequencies for Mica2 motes: uncomment the line which
# has the frequency that you want your motes to use
#Channel 0:
#PFLAGS += -DCC1K_DEF_FREQ=433002000
#Channel 2:
#PFLAGS += -DCC1K_DEF_FREQ=433616400
#Channel 4:
PFLAGS += -DCC1K_DEF_FREQ=434107920
#Channel 6:
#PFLAGS += -DCC1K_DEF_FREQ=434845141

# channel 0-2: 614400
# channel 2-4: 491520
# channel 4-6: 737221
#PFLAGS += -DCC1K_DEF_PRESET=4


# Define for your local group in hex (otherwise it is 0x7d).
DEFAULT_LOCAL_GROUP=0x07

# Choose programmer and port. If nothing uncommented, then assumes
MIB500.
#MIB510=/dev/ttyS5
```

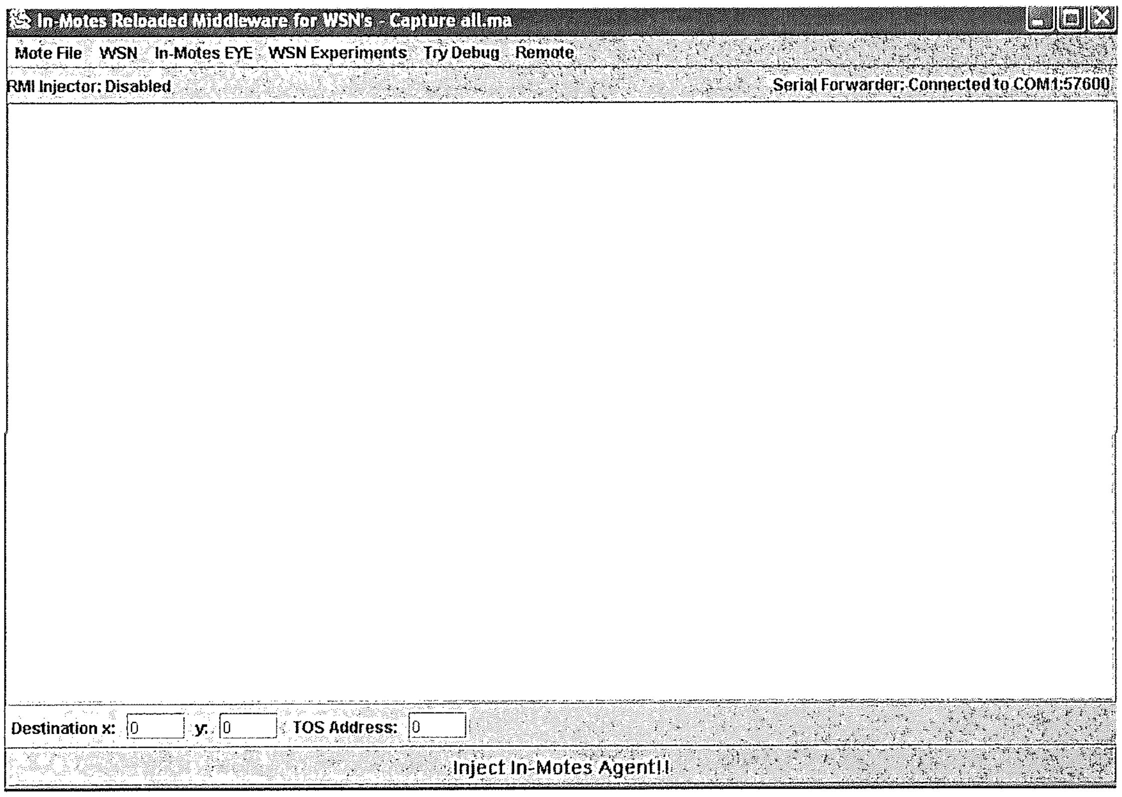
7. Install the In-Motes Reloaded NesC code on every mote, including the base station.
8. Compile the In-Motes Reloaded Java code
9. In a command window specify the TinyOS root /tools/java and type the following command: java -Djava.security.policy=java.policy edu.ee.acnrg.InmotesReloaded.AgentInjector\-comm COM1:57600 -d &

Considering that the portions of In-Motes Reloaded were placed as instructed above and that there were no problems such as exception errors during the compilation of the middleware's Java code, completing the above step us should have the In-Motes Reloaded Java user interface up and running in a new window in your machine. (Take extra care when you setting up the make file so all your sensors in the wireless sensor network are communicating under the same frequency.

The In-Motes Reloaded middleware in pictures

The following section contains a selection of screenshots of the middleware that will explain the main functions of the graphical user interface and will help you to inject your first application in the wireless sensor network

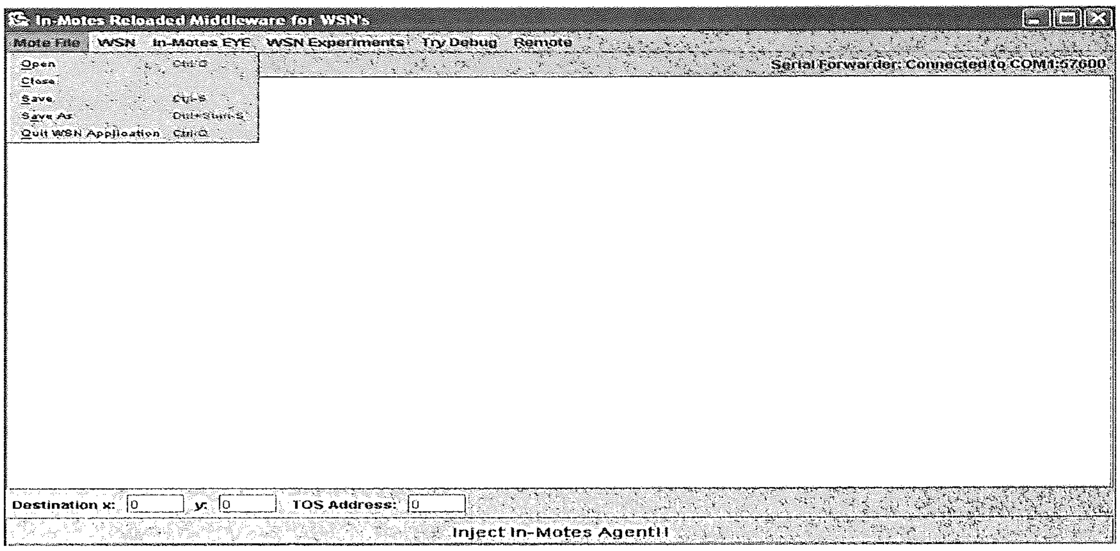
*The **Main Graphical User Interface** of In-Motes Reloaded:*

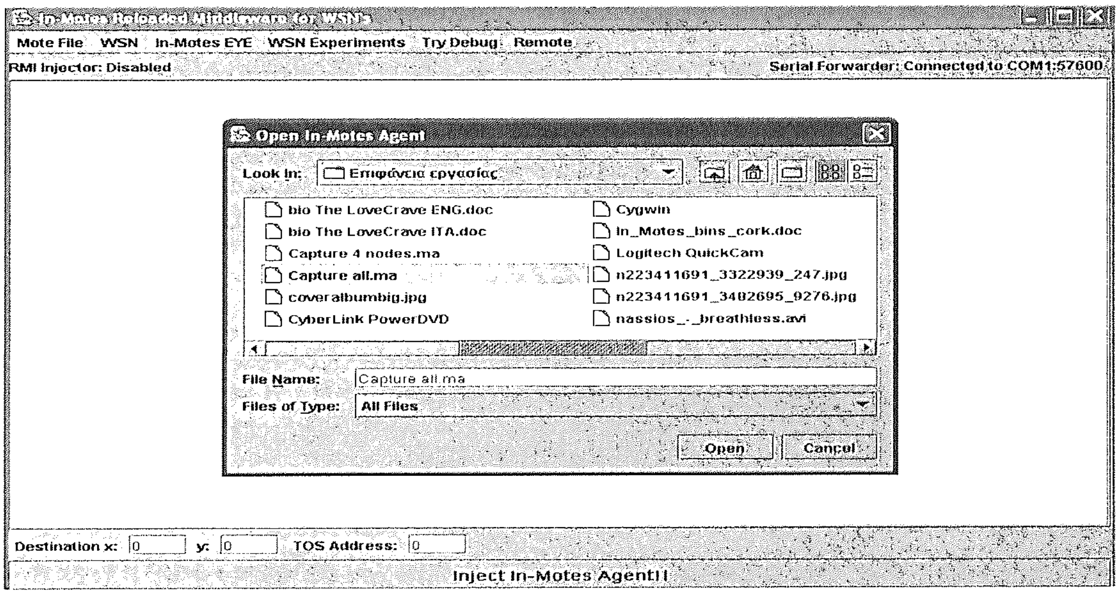


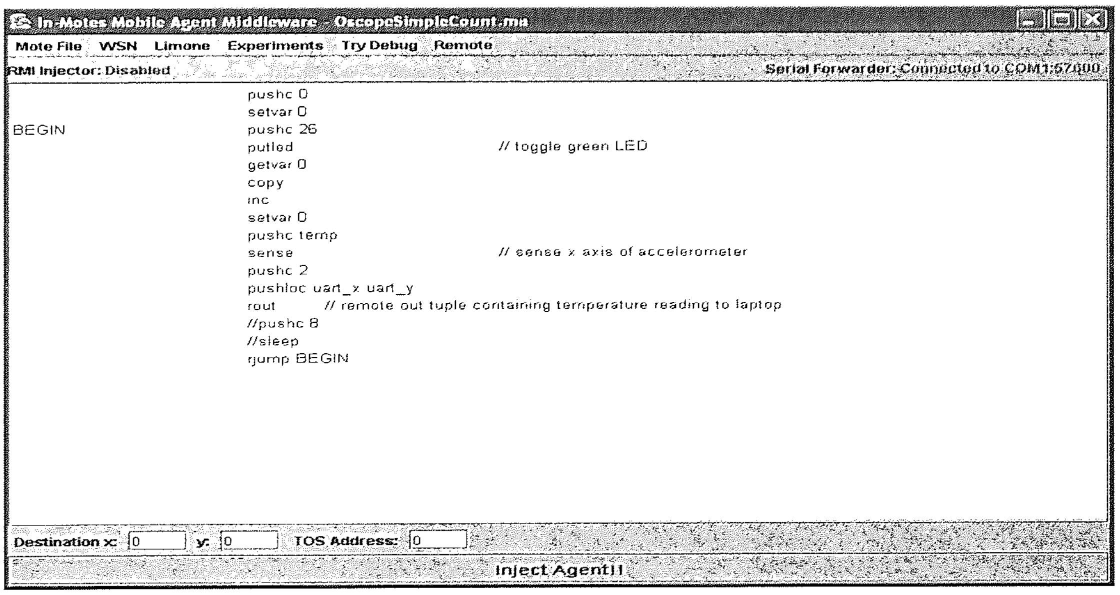
Warning:

Please, unless you are using a grid format for your wireless sensor network do not add anything in the bottom destination boxes. The In-Motes Reloaded engine does not require any pre-specified grid format for the network although it has the ability to generate one. The boxes were added for testing purposes and comparison trials (In-Motes EYE application) and they are purely optional.

*The **Mote File Menu**: Drop down menu allowing a user to open/close/save/save as and quit an application*

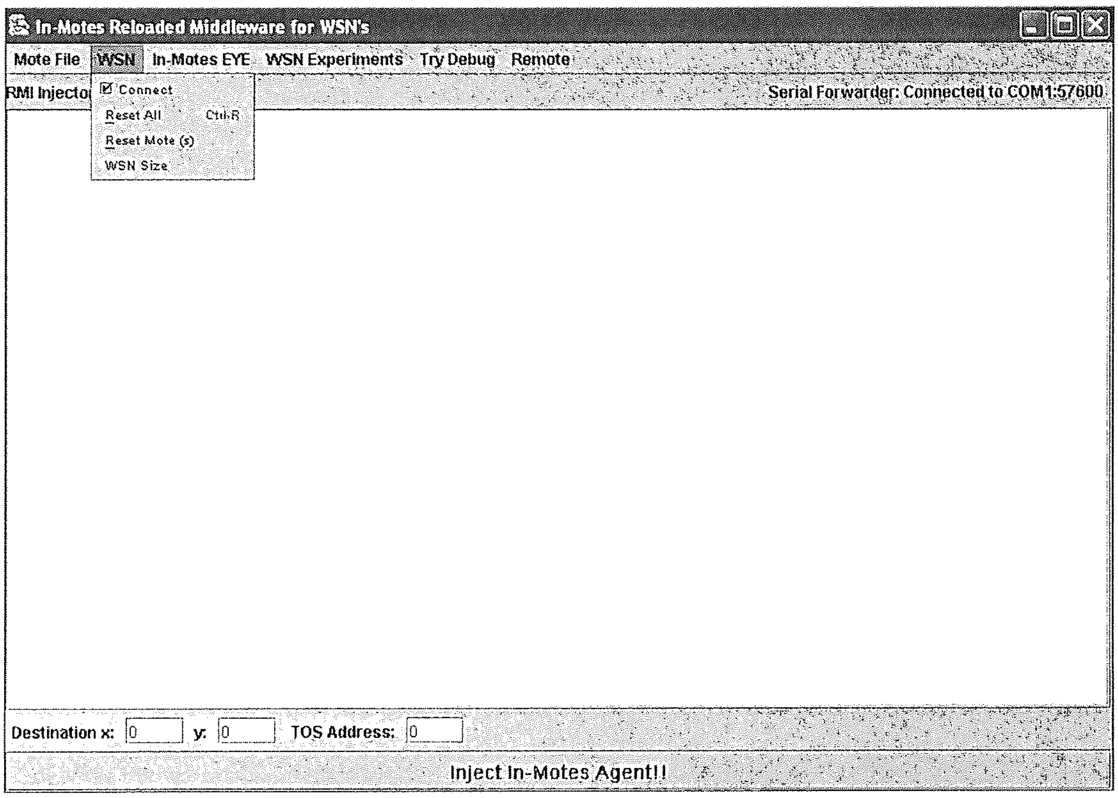




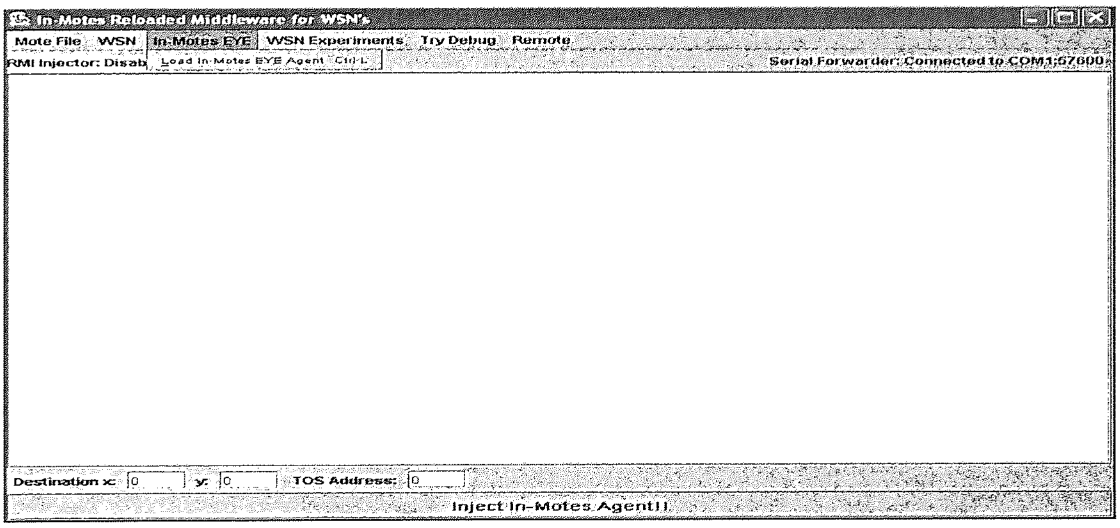


```
                pushc 0
                setvar 0
BEGIN           pushc 26
                putled                  // toggle green LED
                getvar 0
                copy
                inc
                setvar 0
                pushc temp
                sense                   // sense x axis of accelerometer
                pushc 2
                pushloc uart_x uart_y
                rout      // remote out tuple containing temperature reading to laptop
                //pushc 8
                //sleep
                rjump BEGIN
```

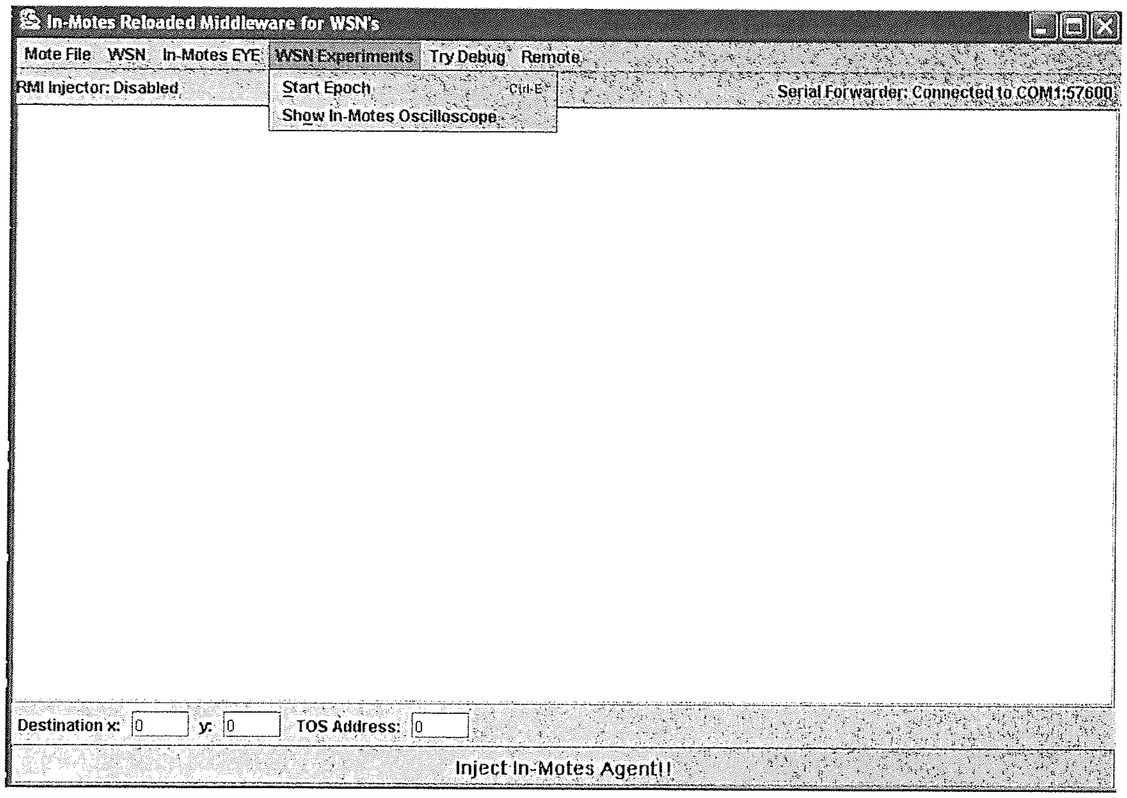
*The **WSN Menu**: Drop down menu allowing a user to connect all/reset all/reset a specified mote in the wireless sensor network or expand or shrink the original network size.*
(Every time you use one of the reset option a Fix Agent is released that will flush the memory of the specified motes of the wireless sensor network)



*The **In-Motes EYE Menu**: Drop down menu that automatically opens and installs the In-Motes EYE application.*



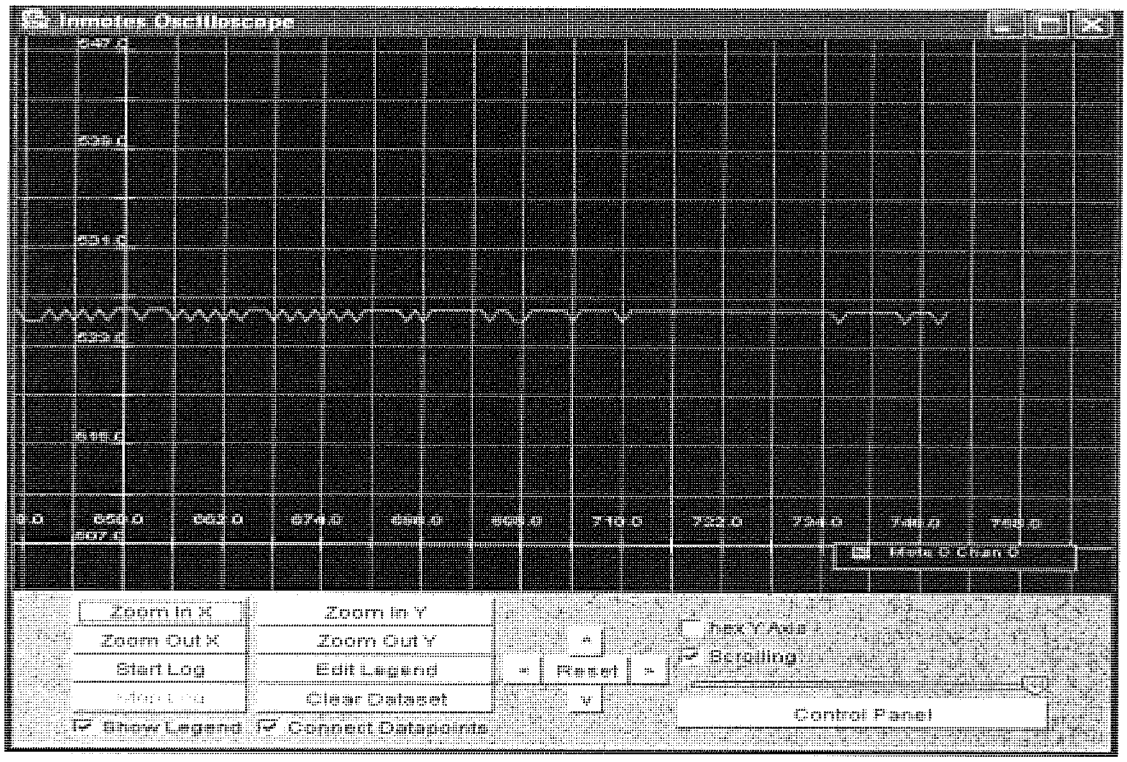
*The **WSN Experiments Menu**: Drop down menu that can start a pre-specified epoch for all the connected motes of the sensor network, assuming that already an application is been injected, it starts also the In-Motes Oscilloscope Interface.*



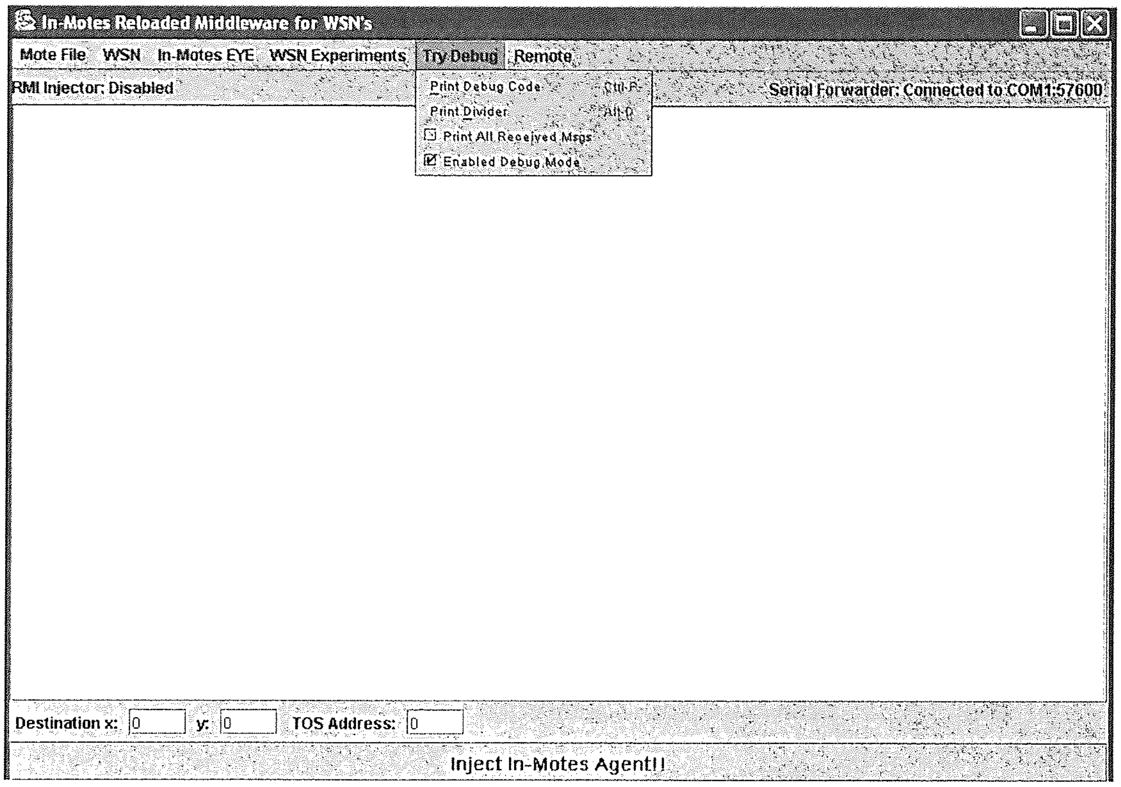
RMI Injector: Disabled

Serial Forwarder: Connected to COM1:57600

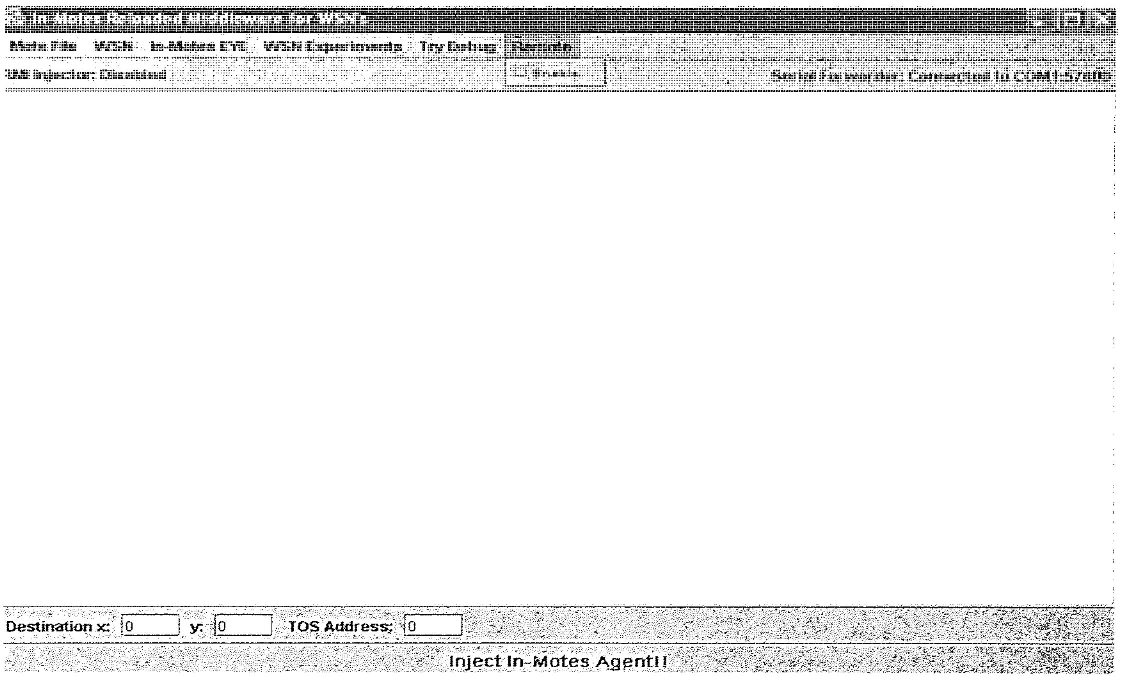
Destination x: 0 y: 0 TOS Address: 0



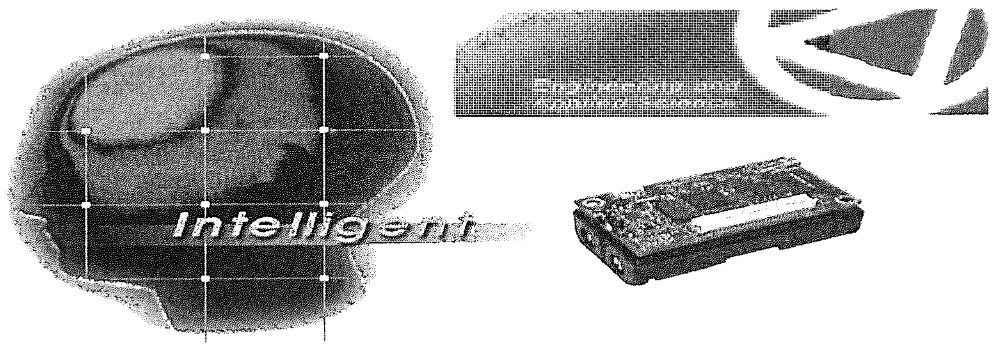
The ***Try Debug Menu****: Drop down menu containing all the options for debugging the middleware or a specified application in real time. If selected, all the additional debugging information will be displayed in the dos command prompt window.*



The ***Remote Menu****: Drop down menu that if selected will allow a user to remotely inject agents in the wireless sensor network using the java remote method invocation. (The user must have first download and install download java.policy and install it in /tools/java TinyOS root directory)*



In-Motes User Guide



In-Motes: An Intelligent Agend Based Middleware for Wireless Sensor Networks

ASTON UNIVERSITY
IN THE HEART OF BIRMINGHAM

Installing In-Motes middleware in your computer

(We strongly recommend to install In-Motes Reloaded as it is a far more stable and energy efficient platform for deploying application in the wireless sensor network. Applications of the In-Motes middleware are compatible with the In-Motes Reloaded middleware)

The following steps will install the In-Motes middleware in your computer. The middleware is applicable for all the versions of the TinyOS operating system and was tested using Windows XP and earlier versions. A user should not have any problems with Linux systems but in the case of Windows Vista make sure that you have set yourself as the administrator of the machine you are using otherwise access problems will occur.

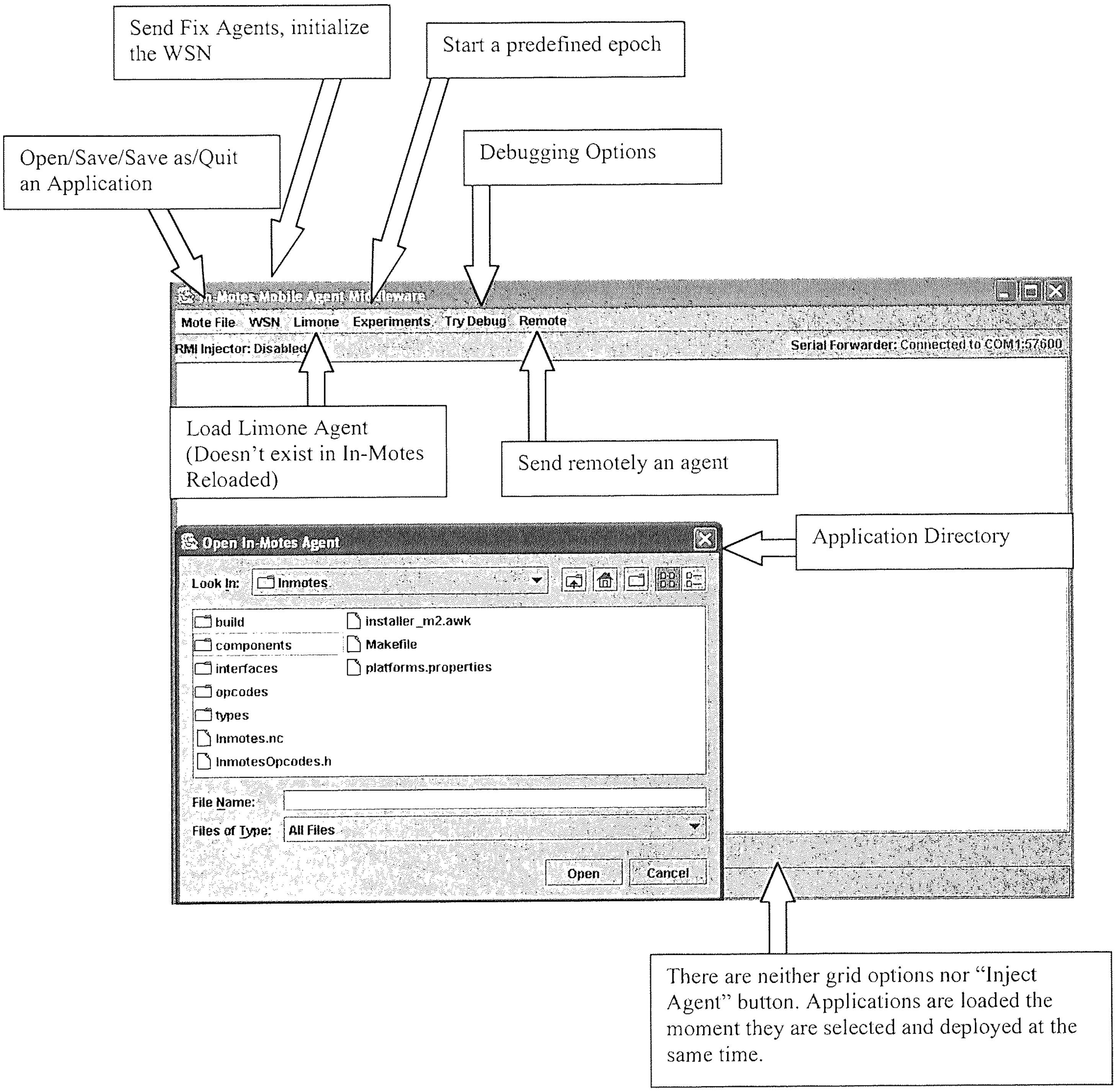
10. Install TinyOS operating system in your computer
11. In-Motes comes as two separate packages with a portion of code in NesC and a second one in Java. Unzip both of them in an area of your computer. They exist in the In-Motes Reloaded folder of the accompanied CD.
12. Create a folder myApps in the TinyOS root path /opt/tinyos-1.x
13. Copy/Paste the portion of the NesC code and add it to the myApps folder
14. Copy/Paste the portion of the Java code and add it to the root path /tools/java
15. Download the makelocal file (displayed below) and install it in /tools/make. Customize the radio frequency, group address and serial port number defined within it. The group address should be unique to you. The serial port number is the port that your mote programming board is connected to.

```
# Radio Frequencies for Mica2 motes: uncomment the line which
# has the frequency that you want your motes to use
#Channel 0:
#PFLAGS += -DCC1K_DEF_FREQ=433002000
#Channel 2:
#PFLAGS += -DCC1K_DEF_FREQ=433616400
#Channel 4:
PFLAGS += -DCC1K_DEF_FREQ=434107920
#Channel 6:
#PFLAGS += -DCC1K_DEF_FREQ=434845141

# channel 0-2: 614400
# channel 2-4: 491520
# channel 4-6: 737221
#PFLAGS += -DCC1K_DEF_PRESET=4


# Define for your local group in hex (otherwise it is 0x7d).
DEFAULT_LOCAL_GROUP=0x07

# Choose programmer and port. If nothing uncommented, then assumes
MIB500.
#MIB510=/dev/ttyS5
```

16. Install the In-Motes NesC code on every mote, including the base station.
17. Compile the In-Motes Java code
18. In a command window specify the TinyOS root /tools/java and type the following command: java -Djava.security.policy=java.policy edu.ee.acnrg.Inmotes.AgentInjector\-comm COM1:57600 -d &

Considering that the portions of In-Motes were placed as instructed above and that there were no problems such as exception errors during the compilation of the middleware's Java code, completing the above step us should have the In-Motes Java user interface up and running in a new window in your machine.

(Take extra care when you setting up the make file so all your sensors in the wireless sensor network are communicating under the same frequency)



Send Fix Agents, initialize the WSN

Start a predefined epoch

Open/Save/Save as/Quit an Application

Debugging Options

RMI Injector: Disabled

Serial Forwarder: Connected to COM1:57600

Load Limone Agent (Doesn't exist in In-Motes Reloaded)

Send remotely an agent

Application Directory

Look In: Inmotes

File Name:

Files of Type: All Files

There are neither grid options nor "Inject Agent" button. Applications are loaded the moment they are selected and deployed at the same time.

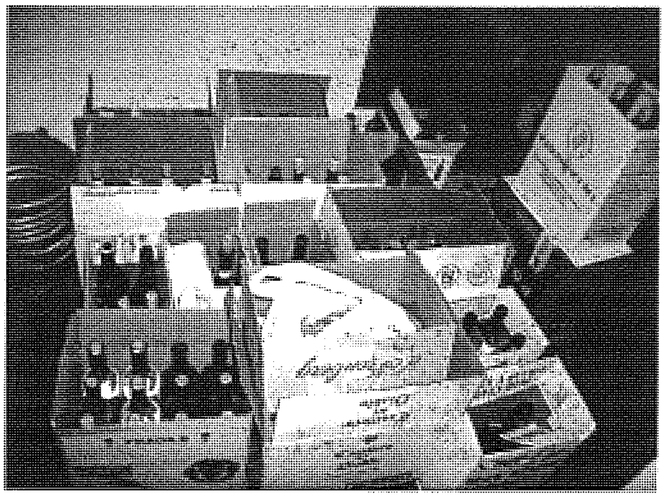
The In-Motes Bins Application

Intelligent
In-Motes Bins: Enviromental monitoring of a Wine Store with the use of Wireless Sensor Networks
oddbins
ASTON UNIVERSITY
IN THE HEART OF BIRMINGHAM
oddbins

- Three main parameters can affect wine: Light, Temperature and Temperature Variations
- We had incidents of corked wines in the sections of France and Italy
- Not everyone brings the corked wine back, we have an average figure of 10 bottles per month
- Customer service is the aim, we try to minimize the number of corked wines as possible
- We believe that In-Motes Bins Application will help us on that extend

Mr. James Gormley

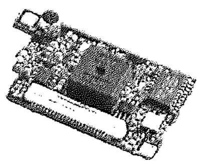
The Problem of Corked Wines:

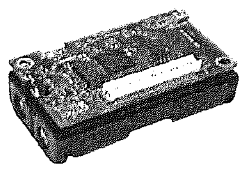


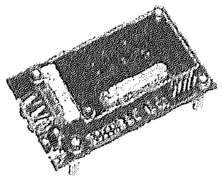
The Solution:

HARDWARE

10 MTS310CA, 11 MICA2 and 1 MIB520CA

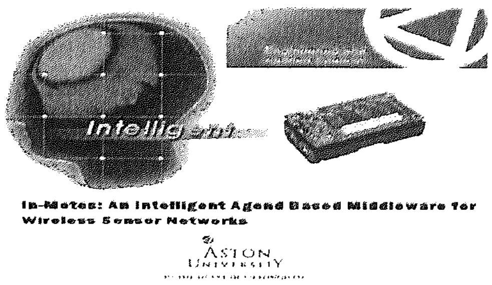






SOFTWARE:

TinyOS

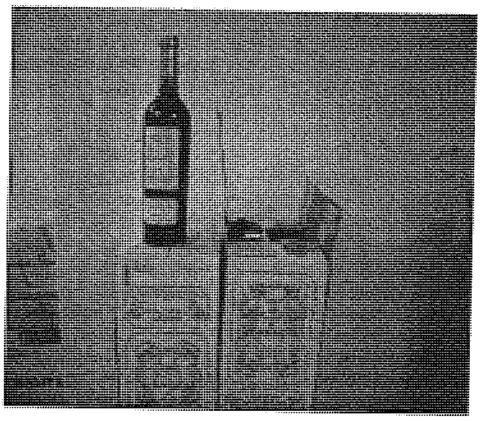


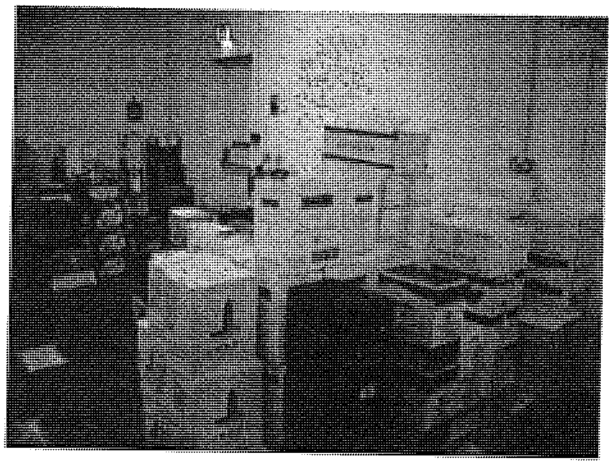
In-Motes: An Intelligent Agend Based Middleware for Wireless Sensor Networks

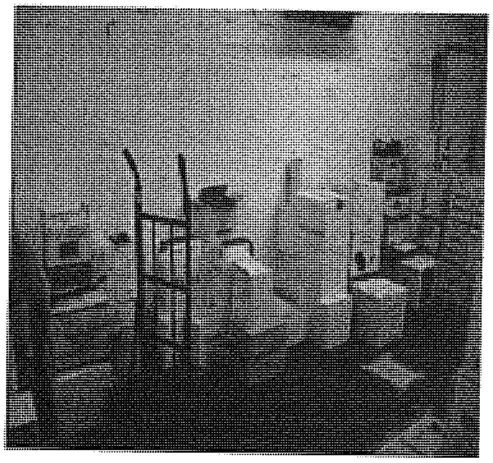
ASTON UNIVERSITY

The Location:

Warehouse:

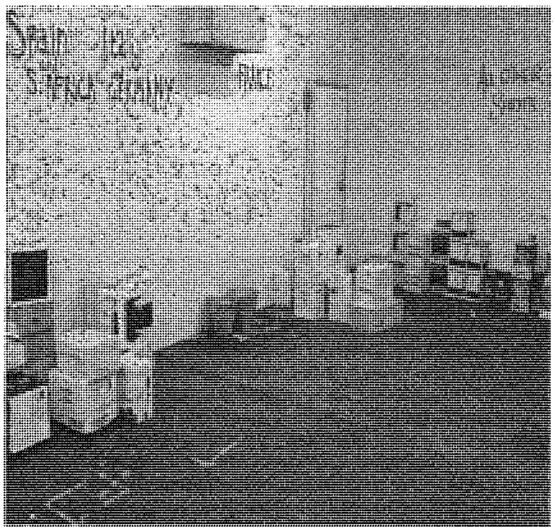


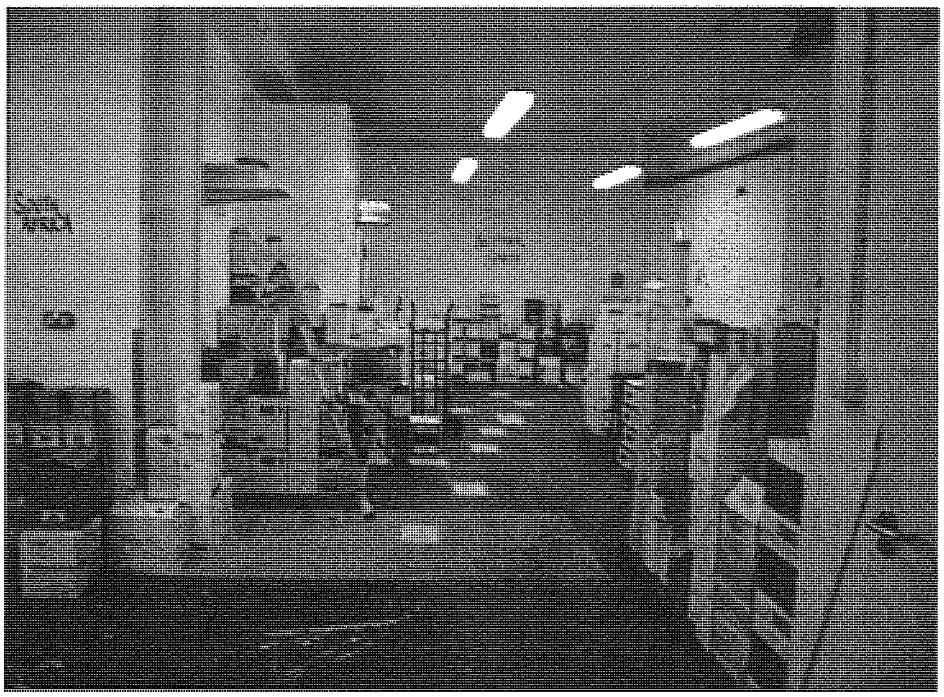


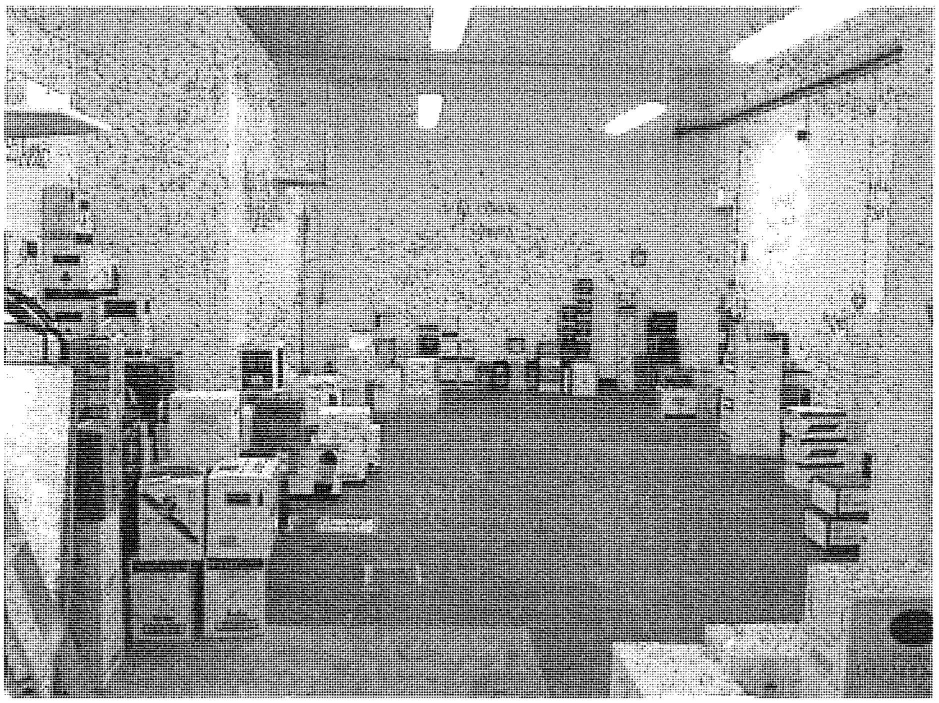


Main floor:

Before and After the trial:







Appendix B