

AD-HOC AND MULTI-HOP WIRELESS SENSOR NETWORKS FOR ACTIVITY
CAPTURE IN COOPERATIVE ROBOTICS

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

Arun Raghunathan

Certificate of Approval:

Ramesh Ramadoss
Assistant Professor
Electrical and Computer Engineering

Thaddeus Roppel, Chair
Associate Professor
Electrical and Computer Engineering

Paul Swamidass
Professor
Management

Stephen L. McFarland
Dean
Graduate School

AD-HOC AND MULTI-HOP WIRELESS SENSOR NETWORKS FOR ACTIVITY
CAPTURE IN COOPERATIVE ROBOTICS

Arun Raghunathan

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirement for the

Degree of

Master of Science

Auburn, Alabama
December 15, 2006

AD-HOC AND MULTI-HOP WIRELESS SENSOR NETWORKS FOR ACTIVITY
CAPTURE IN COOPERATIVE ROBOTICS

Arun Raghunathan

Permission is granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Signature of Author

Date of Graduation

THESIS ABSTRACT

AD-HOC AND MULTI-HOP WIRELESS SENSOR NETWORKS FOR ACTIVITY CAPTURE IN COOPERATIVE ROBOTICS

Arun Raghunathan

Master of Science, December 15, 2006
(B.E., Bharathidasan University, India 2003)

106 Typed Pages

Directed by Thaddeus A. Roppel

Cooperative robotics has been the focus of attention of various groups in recent years. The idea of using a team of robots instead of a single one to execute a task came from the necessity of accomplishing a task that is too difficult or too complex for a single robot. In some situations, using a group of simple robots can be more efficient, less expensive, and more fault-tolerant than having a single powerful, highly specialized robot for each task. Constant, reliable and stable communication amongst the robots and between the robots and a base station becomes highly necessary and useful.

This thesis presents an algorithm for ad-hoc wireless communication among mobile, cooperative robots. Software has been developed to enable the robots to communicate with each other and with a base station. The network is designed to accommodate the failure of any node(s), and the replacement or addition of new nodes as well.

In one operating mode, all communication packets are routed to the base station in addition to their intended destination. In this mode, the base station performs real-time event logging. This enables researchers to replay an entire scenario step-by-step to perform any desired analysis of system operation.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Dr. Thaddeus Roppel for his constant support and encouragement throughout this research. I am also deeply indebted to Dr. Paul Swamidass for his constant support and advice over the last two years. I would like to extend my thanks to Dr. Ramesh Ramadoss for serving on my committee and for his valuable suggestions and comments. A special word of thanks to all the members of the Cooperative Robotics Research team, especially Aaron Steiner, Adam Ray and Rama Narendran, for their encouragement and support. I would also like to thank my other friends at Auburn University for providing encouragement when needed.

Finally, I would like to thank my parents for their constant support and the numerous sacrifices they have made over the years.

Style manual or journal format used: Institute of Electrical and Electronics Journal style

Computer software used: Tiny OS, Microsoft Office Word 2002, Perl, Microsoft Office PowerPoint 2002.

TABLE OF CONTENTS

LIST OF ACRONYMS.....	x
LIST OF FIGURES	xi
LIST OF TABLES	xiii
CHAPTER ONE: INTRODUCTION	1
1.1 Background	1
1.2 Statement of Problem	3
1.3 Outline of Thesis	4
CHAPTER TWO: LITREATURE REVIEW	5
2.1 Instances of Cooperative Robotics	6
2.1.1 Simulation Platform for Cooperative Robotics	10
2.1.2 Communication in Cooperative Robotics	11
2.2 Wireless Sensor Networks	12
2.3 Sensing Cooperative Robotics	14
2.4 Contribution of this Thesis.....	16
CHAPTER THREE: TINY OS AND WIRELESS SENSOR NETWORKS	18
3.1 Wireless Sensor Networks	18
3.2 Tiny OS	20
3.2.1 Concurrency	21
3.2.1.1 Event Driven Execution	21

3.2.2 Modularity	22
3.2.3 Tiny OS Components and Message Structure	23
3.2.3.1 Message Structure	24
3.3 TOSSIM	27
3.3.1 TinyViz	27
CHAPTER FOUR: AD-HOC AND MULTI-HOP ALGORITHM	28
4.1 Scenario	28
4.2 Assumptions	30
4.3 Algorithm	32
4.3.1 Initialization Phase	32
4.3.2 Network Formation Phase	33
4.3.3 Data Collection and Forwarding Phase	34
4.4 Results and Discussions	35
4.4.1 User Interface Discussion	47
CHAPTER FIVE: CONCLUSIONS	51
5.1 Suggestions for Future Work	52
REFERENCES	53
APPENDICES	57
APPENDIX A: CODE IMPLEMENTAION OF AMH ALGORITHM	58
APPENDIX B: JAVA AND PERL CODES	72

LIST OF ACRONYMS

ACK	Acknowledge
AM	Active Message
ADC	Analog to Digital Converter
AMH	Ad-hoc Multi-hop
CRC	Cyclic Redundancy Check
CRR	Cooperative Robotics Research
CSMA	Carrier Sense Multiple Access
GUI	Graphic User Interface
NesC	Network Embedded System C
RFM	Radio Frequency Module
SARA-1	Search and Rescue Algorithm, version 1
Tiny OS	Tiny Microthreading Operating System
TOSSIM	TinyOS Simulator
USB	Universal Serial Bus
VLSI	Very Large Scale of Integration
WSN	Wireless Sensor Network

LIST OF FIGURES

Fig. 1: Example displaying various components in Tiny OS.	23
Fig. 2: A screenshot of the TinyViz software simulating the multi-hop protocol implemented on 90 sensor nodes.....	26
Fig. 3: Flowchart of the AMH algorithm.	31
Fig. 4: Experimental Setup.	32
Fig. 5: A graphical representation of the initial setup of the network.	38
Fig. 6: A screenshot of the readings at the base station during the initial phase of the network.	39
Fig. 7: A graphical representation of Node 3 breaking down.....	40
Fig. 8: A graphical representation of Node 6 requesting a new parent... ..	40
Fig. 9: A graphical representation of Node 5 responding to Node 6 with an “I am a Parent” signal.	41
Fig. 10: A graphical representation of the changed parent of Node 6.	41
Fig. 11: Screenshot displaying the changed parent of Node 6. The parent was changed from Node 3 to Node 5.....	42
Fig. 12: A graphical representation of Node 3 re-entering the network.....	44
Fig. 13: A graphical representation of Node 5 responding to Node 3’s request.....	44
Fig. 14: A graphical representation of the seamless re-entry of Node 3 into the network.	45
Fig. 15: Screen shot displaying the seamless reentry of Node 3 into the network....	46
Fig. 16: A screenshot of the text file of the information recorded at the base station.	48

Fig. 17: A screenshot showing the upper half of the output file.49

Fig. 18: A screenshot showing the bottom part of the output file.50

LIST OF TABLES

Table 1: Table showing various fields in a TOS_Msg data structure.	25
---	----

CHAPTER ONE

INTRODUCTION

1.1 Background

The field of multi-robot systems is a logical extension of the research on single robot platforms. Single robots, however efficient and effective, are restricted in the area they can cover at any given time. In the place of a single, expensive, complex robot, multiple cooperating robots with lesser capabilities and features can sometimes be made to do the same task more efficiently.

Multi-robot systems also differ from other common distributed systems such as computers, databases and networks because of the relative difficulty in modeling the environment in which they operate [9]. Cooperative robots generally interact amongst themselves, and at the same time there is a constant need for these robots to sense and react to the environment in which they are functioning. Hence to have a successful cooperative robotics setup, it is essential to have a good understanding of the environment in which the robots are going to operate.

Cooperative Robotics has come a long way since Grey Walter, along with Wiener and Shannon, in the mid 1940s analyzed turtle-like robots fitted with light and touch sensors; the scientists observed that the uncomplicated robots displayed a “complex

social behavior” in responding to each other’s movements [14]. Significant research within the field of cooperative robotics started in the 1980s. Initial projects such as SWARM [7], ACTRESS [2], and CEBOT [16], however were limited to just simulation work. Research in three distinct task domains pushed the field of cooperative robotics into the physical implementation arena. These were:

- i) Traffic Control: Traffic control issues inspired researchers to look into solutions for collision avoidance and communication protocols in multi robot scenarios.
- ii) Cooperative Manipulation: Cooperative manipulation issues encouraged researchers to look at distributing task allocation and fault tolerance.
- iii) Foraging: Example foraging applications include search and rescue missions and hazardous chemical cleanup. A single robot may be programmed to do these tasks. Using multiple, cooperating robots allows us to study the improvement in performance achieved over using a single robot.

Present day research scenarios for cooperative robots include search and rescue, remote surveillance, security systems, and landmine detection and removal.

1.2 Statement of Problem

In this thesis we present an algorithm that facilitates the wireless communication between mobile, cooperating robots. The algorithm enables the robots to be a part of a self-forming network. Initially when they start out, the robots do not have an awareness of other robots or the information to gather. Eventually with the help of a beacon from

the base station, the robots arrange themselves into a hierarchical network setup with varying node depths. Once the network formation is complete, nodes start to collect information from the surroundings and start communicating with each other. A copy of the message exchanged is also sent to the base station. Since communicating robots may be out of direct communication range with the base station, the algorithm makes provisions for the messages to multi-hop through fellow robots, back to the base station. The base station apart from initiating network formation also performs real-time event logging. This can be useful to scientists who may want to observe the sequence of events happening in the network. Additionally, during the case of a node failure or during the addition of a new node to the existing setup, the network is designed to be flexible to accommodate such changes.

The wireless communication algorithm is implemented on a T-Mote Sky [40] platform using Tiny OS [36]; however it can be extended trivially to suit the MicaZ [25] or the Mica2 [24] platforms. The above platforms are examples of fundamental units in a Wireless Sensor Network. A basic unit of a Wireless Sensor Network (WSN) typically consists of two to three sensors, a microcontroller and an extremely low power consuming radio transceiver. We have chosen wireless sensor nodes to act as the communication module for our cooperating robots because they serve the dual purpose of sensing the environment as well as enabling low power communication between other participating robots.

The algorithm is realized in software using the Tiny OS operating system that was created specifically for developing applications in the WSN domain. Tiny OS is different from other embedded operating systems in that it acts more as a framework to download

an application-specific operating system into individual nodes, rather than as a traditional kernel-based OS. Tiny OS and WSN are described in more detail in Chapter Three of this thesis.

1.3 Outline of Thesis

The remainder of this thesis is organized as follows. Chapter Two presents an overview of the existing literature in the field of cooperative robotics. More specifically it delves into the literature that describes the use of sensor fusion and other sensing techniques applied to cooperative robotics. Chapter three gives an overview of the Tiny OS operating system, and a description of the experimental setup. Chapter four is devoted to a description of the ad-hoc and multi-hop routing algorithm (AMH) and the results observed. Finally Chapter five presents the conclusion, summary and suggestions for future work.

CHAPTER TWO

LITERATURE REVIEW

The field of cooperative robotics has witnessed significant developments over the last fifteen years. Research in this field has diversified into several realms extending from human-robot interaction to object finding and terrain mapping. Due to its developments in various fields, a single, explicit definition does not exist. Of the many definitions that exist, the following are found most relevant:

- 1) “joint collaborative behavior that is directed toward some goal in which there is a common interest or reward.”[4]
- 2) “a form of interaction usually based on communication”[22]
- 3) “[joining] together for doing something that creates a progressive result such as increasing performance or saving time.” [29]; and finally
- 4) Given some task specified by a designer, a multi-robot system displays cooperative behavior if, due to some underlying mechanism (i.e., the “mechanism of cooperation”), there is an increase in the total utility of the system.[9]

This thesis is devoted to combining the field of WSN and cooperative robotics. The initial part of this chapter describes the developments of sensor fusion techniques for cooperative robotics. We then proceed to describe some routing techniques that are used for communication in WSN and cooperative robotics. These routing techniques were

used as an inspiration for the Ad-hoc, Multi-hop (AMH) algorithm described in this thesis. We finally conclude the chapter with a description of the Cooperative Robotics Research (CRR) at Auburn University and the contribution of this thesis towards the same.

2.1 Instances of Cooperative Robotics

Multiple robots working together in coordination with each other has several advantages compared to single robot solutions. These include better performance, increased fault tolerance, and distributed sensing and actuation. Multiple robots working in teams can perform tasks accomplished by a single robot using a simpler design and at a quicker pace. In the following paragraphs, we describe various instances of research that are being currently undertaken in the field of cooperative robotics

Carlson et al. [10] have developed a system of four robots that have been built to perform search and rescue missions. Robots working in tandem search a given area and locate victims. The robots apart from finding the target also build maps using sensors to help human search and rescue teams. The robot groups also implements an ad-hoc network protocol that enables them to communicate with each other when their existing network infrastructure has been damaged or is unavailable in certain areas. The working of the robot group was simulated in a real-time environment before being implemented in hardware.

Almeida et al. [1] describes another group of five robots that were built for the purpose of taking part in the RoboCup soccer competition. This group of robots is highly

sophisticated in design and is capable of achieving a high degree of coordination. This coordination is achieved through the use of two webcams functioning as sensors. One webcam has an omni-directional vision system and is used for obstacle avoidance and ball handling. The other camera is used for monitoring the ball over medium and long distances as well as track other players and goal posts. On the software front, a PC based node is used as a central controller that facilitates the interconnection between various devices on board such as the camera, motors etc. through standard interfaces such as USB, Firewire etc. Each robot runs a Linux operating system on the central PC node.

Chaimowicz et al. [12] propose an architecture for communication among cooperating robots. The proposed architecture is suitable for multiple robots performing tasks that cannot be accomplished by a single robot. In such cases there is a high level of coordination required among the robots to get the task done. The main feature of the architecture proposed by Chaimowicz et al. is a role assignment mechanism that enables reconfiguration of the participating robots. A robot that serves as the leader of the group may become a follower at any time and conversely a follower robot may become a leader of the team. In an ideal situation a leader plans the path of the group and broadcasts it to its followers who control their trajectory according to the information received and the information received from their own sensors. The Chaimowicz et al. architecture with its ability to reconfigure robots enables them to be more adaptive and robust. Since the robots are capable of exchanging roles, they can adapt themselves better to suit the task at hand.

Simmons et al. [34] address the problem of mapping and exploration of an unknown environment using a group of robots. Mapping and exploration is an area where

the advantage realized through multiple robots is compensated by the interference occurring among the robots. As a result, two robots might map the same area and this leads to a reduction in productivity of the group. The algorithm [34] provides a way to coordinate the movement of multiple, heterogeneous robots during their task of mapping and exploration. The mapping problem is tackled by allowing each robot to build a local map of its area and pass it to a central module. The central module then integrates these maps to come up with a global map of the entire area. Once this done, each robot submits a “bid” that consists of the cost of each robot to explore a particular area. The central module once again determines which robot should explore which area according to the “bids” submitted.

A large, mobile robot team capable of autonomous functioning was designed and deployed by Howard et al. [20]. The robots are expected to map the interior of a building, deploy a sensor network and use the sensor network to track for any intruders. Eighty robots capable of performing mapping, tracking and exploration were built and deployed. The robots built were of two types; the first kind of robots is more sophisticated, equipped with cameras and powerful CPUs. The second kind of robots (sensor robots) is smaller and equipped with simpler cameras and fewer components. The overall mission is divided into two phases; during the first phase robots explore and map the building’s interior. They also build an occupancy grid map of the building that is used to determine the locations for deploying the sensor robots. When the coordinates are determined, the sensor robots are assisted by the bigger robots to their location and once there, the sensor robots form a distributed sensor network that is used to track intruders.

Mobile robots were/are designed to perform both mundane, routine tasks as well as work in unstructured environments. In such cases it would be convenient if there were robots that can support multiple modalities of sensing, manipulation and locomotion. Rus et al. [32] describe a versatile group of robots capable of reconfiguring themselves to suit the application and environment. They propose to have hundreds of small modules that will automatically reconfigure themselves to suit the terrain and other needs of the given task. The process makes use of mobile sensors that help self configuring robots by providing them a sense of direction. Small robots pushing furniture may not be able to see what is ahead of the object and the mobile sensors act as a guide helping the robots by directing their movements.

Das et al. [13] describe another set of heterogeneous group of robots that can be used for search and rescue missions. In this case, robots are sent in and are responsible for deploying static sensor nodes at various locations inside the building. The sensor nodes, once placed, automatically configure themselves into a distributed sensing platform that provides applications for its consumers pertaining to monitoring and surveillance. Das et al. combine networking and sensing ideas to control the flow of information in search and rescue missions in an unknown environment. They importantly address the issues of localization for sensors and robots in an unknown environment, the process of collecting information from different sensors from different parts of a building and integrating it at a central node to obtain a global map of the building, and finally using feedback from the static sensor network to control the movement of autonomous robots inside a burning building.

Fontan et al. [33] address one of the key issues in cooperative robotics; interference amongst participating robots. They (Fontan et al.) claim that minimizing interference amongst participating robots and increasing the synergy between them is the best way to realize adaptive behavior in multiple robot scenarios. The paper proposes a multi-robot controller that is inspired by behavior-based approach [23]. Using this controller, each robot is assigned a particular region of the terrain that is supposed to be cleaned. When one robot malfunctions or breaks down, its territorial area can be resized on the fly so as to not affect the productivity of the robot group.

Finally, Parker et al. summarize the current state of technology in the field of autonomous mobile robots in [27]. They describe the state of research in eight topics within multi-robot systems: biological inspirations, communication, architectures, localization/mapping/exploration, object transport and manipulation, motion coordination, reconfigurable robots and learning in multi-agent systems.

2.1.1 Simulation Platforms for Cooperative Robotics

Turnell et al. [37] discuss a simulation tool called SIMBOT to design and implement autonomous robots in a cost-efficient manner. The tool allows research groups to create an environment containing multiple robots and develop programs to control their behavior. The tool has provisions for physical modeling of the scenario and simulates other devices such as cameras for producing 3-D views of the environment for the users. SIMBOT also provides a host of other components (robot parts and a “brain”) to speed up the design implementation in virtual environments. SIMBOT provides an

ideal platform to test critical concepts in mobile autonomous robots before being implemented in hardware.

Gerkey et al. [17] describe the use of a software called Player that acts as a network interface to sensors and actuators on various robots. Player allows the client to access sensors and robots on multiple robots in different parts of a single network. Player allows a convenient method to access robot data at the client by providing abstract interface to various devices on board. These devices can be programmed to send data back to users at regular intervals. Player provides an excellent interface to remotely collect and display data from robots and sensors. Stage is a GUI platform that can be interfaced with Player to simulate the behavior of multiple robots in a given environment. An algorithm for a search and rescue mission, SARA-1 [30], was implemented by Adam Ray using Player/ Stage platform.

2.1.2 Communication in Cooperative Robotics

Balch et al. [3] analyze the importance of communication in a cooperative robotics setup through a series of experiments performed on both simulated and real robots. The performance of the robot system was monitored for three different tasks with three different types of communications. The level of communication is progressively increased to be more complex and expensive for each task. It was observed that for some tasks, communication significantly improves performance while for others; it increased the cost of implementation and was found unnecessary. In situations where

communication between robots helped improve performance, the simple form of communication was found to be as effective as the more complex ones.

Hustin et al. [21] describe a communication strategy for cooperative robots. This strategy includes a new language and a protocol for communication between mobile robots. The authors assume that inter-robot communication occurs over a high frequency RF channel. The language and protocol proposed are designed to operate in this band and provide safe communication between multiple robots. The communication between the robots is done using a radio accessed by CSMA technique. The protocol ensures a highly reliable communication by using acknowledgement and time-out processes. The language described also employs techniques such as macro messages to reduce network congestion. Macro messages are used to reduce the size and number of messages.

2.2 Wireless Sensor Networks

In the recent years there has been a tremendous growth in the fields of wireless networking, VLSI, and embedded processors. This has led to the emergence of the whole new domain of WSN. Tiny sensors may be dispersed into the air, roads, walls, or machines creating a digital skin that can perform a variety of functions: monitor pedestrian and vehicular traffic, report wildlife habitats, aid rapid emergency responses, and track inventory flow in factories [39]. In this section we summarize the literature relevant to the applications of WSN. In the next section we describe the current state of research in using WSN for mobile, cooperative robots.

Estrin et al. [11] discuss habitat monitoring as one of the applications used to drive the development in WSN technology. The aim of performing a habitat monitoring exercise is to understand the biocomplexity in various environments. To perform the biocomplexity mapping, we require sensing systems that are able to configure themselves to adapt to the dynamic environment. The sensing system should be able to operate for long periods of time with fixed amounts of energy. In other words, there should be an inherent technique for energy conservation in these systems. WSN satisfy all these requirements and provide an ideal platform to be dispersed in any environment for habitat monitoring.

WSN's have found applications in patient monitoring and care too. In Bauer et al. [6], a patient monitoring and care system with three layer distributed sensor network is described. The three layers consist of a leaf node, an intermediate node layer and finally a root node present at the central monitoring facility. The most basic layer is the leaf node which is a collection of sensors monitoring critical aspects of the patient such as blood pressure etc. The intermediate level nodes are represented by more intelligent processors also carried by the patients, and are responsible for distributing the available leaf sensors according to the criticality that is handed down by the root node at the monitoring facility. The intermediate node collects the information from individual node sensors, processes it, and sends it to the root node at the hospital/monitoring station.

Peterson et al. [28] discuss the development of a distributed algorithm to interact with a sensor network. The objective of the distributed algorithm is to guide a robot/human through a region with the aid of multiple sensors. A sensor network deployed in the region initially models the area, adapts to any changes, and guides any

moving object incrementally through the area. The authors also propose the use of an external device to interact with the sensor network. This device is responsible for collecting navigational information from the sensor in the neighborhood, activating and deactivating specific areas of the sensor network and detecting events occurring in the sensor network.

2.3 Sensing in Cooperative Robotics

Sensing is a fundamental aspect of cooperative robotics. Sensing enables robots to have an idea of localization and gives them an opportunity to interact with the surrounding environment. Modern day robots usually have different sensor systems with different sensing modalities and different characteristics on board. Values from these sensors are combined and processed to provide information that is more accurate and robust compared to values from a single sensor reading. This is known as sensor fusion, and is widely employed in present day cooperative robotics.

Pereira et al. [27] describe a distributed sensing mechanism for cooperative robotics. In this system the robots are allowed to share their information with each other. This system is different from [12] where a robot cannot obtain the sensor readings of other robots, but receives only processed information based on the sensor readings. In the proposed system, any robot in the team may use the data from sensors on any other robot to aid localization or to enable the execution of cooperative tasks. For this to happen, each robot must send out its readings to all other robots. The other robots, based on a

predefined criterion, decide which data is relevant. They then fuse the obtained data with their own set of information and execute the given task more efficiently.

Burhanpurkar describes a real world application for a high-performance sensor system for use in mobile robots [8]. Navigation systems and path planning system for autonomous robots generally require sensors that are capable of precisely detecting and differentiating objects of various sizes and shapes. Such sensors are usually expensive and of limited value in the real world. The author proposes a new sensor system comprising of a modular ultrasonic array that is capable of delivering a low cost solution for achieving high performance in a dynamic, unstructured environment where autonomous mobile robots are used.

Stroupe et al. [35] present a method for fusing noisy and uncertain observations of an object made by multiple robots. The fusion of sensory data allows accurate modeling of the environment and increases the effective visibility of the robot group. Furthermore the authors have proved using their approach that when there are more observers there is a more accurate estimation of an object's location. Along the same lines, Wijk et al. [38] have proved that a grid based sensor technique consisting of multiple observers (sensors) is most useful for sensor fusion, localization, obstacle avoidance and navigation of mobile autonomous robots.

Batalin et al. [5] have proposed one of the first systems to use sensor networks for mobile robot navigation. In this system, a WSN is deployed to map the environment. A mobile robot navigates through the area by constantly communicating with the sensor nodes using a low power radio onboard. The authors have developed an algorithm for computing the closest sensor node based on received signal strengths. A unique aspect of

this system is that the robot does not have to be complicated, as the localization and mapping are performed by the embedded sensor network.

2.4 Contribution of this Thesis

This thesis proposes a new ad-hoc, multi-hop algorithm (AMH) that can be used for inter-robot communication as well as communication between individual robots and a base station. In the literature reviewed in the previous sections, although mobile sensors were used for guiding robots through unknown environments, they were not mounted on the robots and existed separately. In cases where sensors were mounted on the robots they were merely used for detecting various parameters for the purpose of fusing data and not for communication between them. The wireless sensor node is a unique platform that enables sensing and communication to be embedded into a single platform. According to the author's knowledge, this thesis is one of the first to use WSN for both communication and sensing in cooperative robotics.

The AMH algorithm is inspired by the various ad-hoc routing protocols [31] for wireless networks. However, the existing wireless routing protocols are not built for systems having memory constraints such as the nodes in a WSN. The AMH algorithm is a unique algorithm that exploits the characteristics of a WSN by requiring a very small overhead for communication. It uses the event-driven feature of Tiny OS that requires a node to service a request as soon as it receives it thereby eliminating a large buffer for incoming messages.

The AMH algorithm was developed specifically for cooperative robots being built as part of the CRR group at Auburn University. The key objective of the CRR group is to build robots that can perform tasks in a cooperative fashion and use sensor fusion techniques for enabling better decision making among the robots. The development of the AMH algorithm is a contribution towards this objective.

CHAPTER THREE

TINY OS AND WIRELESS SENSOR NETWORKS

The proposed AMH algorithm has been implemented on a WSN using an operating system called Tiny OS (Tiny Microthreading Operating System). Tiny OS is an operating system developed specifically for wireless embedded sensor networks. The official website of Tiny OS defines Tiny OS as follows:

“Tiny OS features a “component-based” architecture that enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks.”

The component library of Tiny OS includes network and routing protocols, sensor drivers and data acquisition tools.

This chapter presents an overview of the Tiny OS operating system and also the rationale behind using WSN nodes as the communication module for the cooperating robots.

3.1 Wireless Sensor Networks

Each node in a WSN must perform sensing, computation and communication. Recent advances in the field of low power electronics and VLSI have been major factors in reducing the size of these nodes. Current research is directed towards accommodating the capabilities of sensing, computation and communication into a device of 1 cm^3 [19].

The advantage of using WSNs lies in their ability to be deployed in the form of numerous tiny nodes that can assemble themselves into a self-forming network. Existing networks could be incremented by simply adding new nodes and no major reworking or configuration is required. A WSN is different from cell phone systems that deny service when too many phones operate in a given area. When the number of nodes deployed in a particular area is increased, the quality of service obtained from the network may be enhanced. Typical applications of WSN's include real-time tracking, monitoring of environmental conditions and ubiquitous computing environments.

Another advantage that WSN's have over other wireless devices is that they do not need to communicate with a central tower or a base station; it is sufficient if they are able to communicate with neighboring peer nodes. Since WSN nodes adopt a mesh topology, data transfer between any two nodes is conducted through messages that may multi-hop via several nodes before reaching the destination. Thus no preexisting infrastructure is required for maintaining the mesh topology. In other words, WSN are a readymade solution for ad-hoc, self forming networks. This feature also results in significant cost and time savings during network deployment.

Cooperating robots need to be able to communicate with each other about the environment they are operating in. They must be able to configure themselves into a fully-functioning network with minimal help from a central system. Such network formation is critical to effective cooperation amidst participating robots. The ad-hoc nature of sensor networks makes it a logical choice for cooperative robotics.

Robots usually carry on-board multiple sensors to detect the various physical parameters around them. There have been research instances where participating robots

have been programmed to share their sensor readings with each other to collectively create an overall picture at the base station. Wireless sensor nodes, having an innovative design that combines sensing, processing and communication on a single device are a one-stop solution to create an effective communication system amongst cooperating robots.

3.2 Tiny OS

The role of an operating system is to provide a reliable environment for the development of applications to suit the system. Tiny OS was developed in the labs of University of California at Berkeley, specifically for WSN. Tiny OS is an operating system with an extremely small memory footprint and differs from traditional operating systems by not supporting memory management or virtual memory. While designing operating system for WSN, two main issues, concurrency and modularity, need to be addressed.

3.2.1 Concurrency

Processes occurring in a WSN are usually concurrency intensive. That is, there are multiple streams of data flowing through a node that is part of a larger network. On the one hand, a node may be carrying out its normal data acquisition function; there also arises the need to transfer packets or service protocol events that might arise asynchronously from the network. Tiny OS introduces a highly event-driven model that

is component based and provides a high degree of concurrency while supporting the major requirements of small footprint and low power consumption.

3.2.1.1 Event-Driven Execution

In several embedded systems, operating systems handle concurrency-intensive processes by allowing multiple threads for multiple operations. Each of these multiple threads is allocated its own stack and set of registers. Whenever a thread indicates a completion of operation, the registers and state variables are saved during the context switch to another operation. This is possible in complex embedded system with considerable memory availability; however in the case of WSN's with severe memory constraints this technique is not affordable.

Tiny OS implements a smart event-driven scheme that uses a single stack to support concurrent activities. A central dispatcher component calls specific event handlers depending on the event type and state of the system. Handlers that are called execute immediately and update the state of the system upon termination.

3.2.2 Modularity

Another important issue relating to WSN is that each system should support a fair degree of modularity. Tiny OS is composed of several libraries of components. Each component is a well defined object with interfaces and internal concurrency. Components generally encapsulate hardware elements on the wireless sensor node such as ADC,

timers, radio etc. Components also provide commands to the user and signal events when tasks are executed.

Tiny OS is written using NesC (Network Embedded System C), a dialect of ANSI C. Components support a bidirectional command/event interface that is implemented using split-phase operations. During a split-phase operation, a higher level component issues a command, requesting a service to a lower-level hardware device. The command returns immediately to indicate whether the command placed was successful or not. It should be noted however that the requested operation might not have been completed by now. Once the placed request is executed, the hardware component returns a signal to the higher level component signifying the completion of task. Between the two occurrences (placing of command and signal completion of task), the node may enter a sleep mode to conserve power. Thus such an interleaving of execution enables the power management feature of Tiny OS.

3.2.3 Tiny OS Components and Message Structure

The Tiny OS components are of three types: hardware abstractions, synthetic hardware and high level software components.

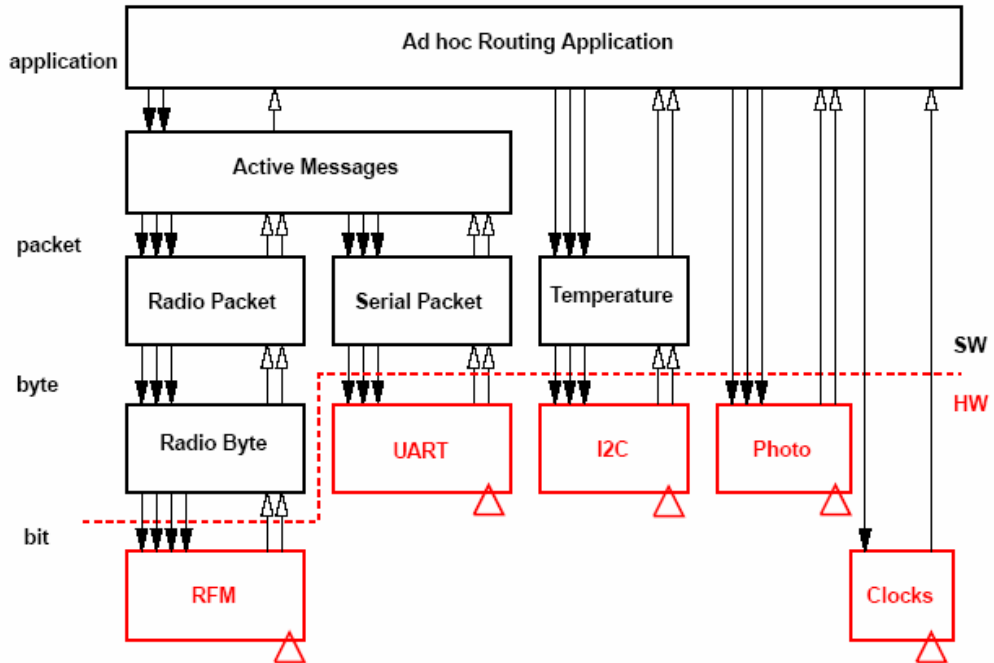


Fig. 1: Example displaying various components in Tiny OS.

Hardware abstraction components are used to map the physical components available on a WSN platform. An example is the RFM component shown in Fig. 1. This component supports commands to manipulate the I/O pins of the radio transceiver and signals an event whenever bits are transmitted or received.

Synthetic hardware components generally simulate the presence of advanced hardware. An example of such a device would be the Radio Byte component in Fig. 1. This component is used for performing basic encryption and decryption of data bits that flow in or out of the RFM hardware abstraction component. In the perspective of a higher

level component, the synthetic hardware component and hardware abstraction component are no different from each other as they provide the same interfaces and functionality. However it provides the user with much needed flexible components while designing algorithms for WSN applications.

The high level software component is responsible for the control and routing of data. An example of this component in Fig. 1 would be the Active Message and Ad-hoc routing application module. These modules are responsible for filling data packets and sending messages to their appropriate destinations.

3.2.3.1 Message Structure

The communication concept in Tiny OS is the Active Message model used in computing systems requiring a high degree of parallelism. In Fig. 1, the application layer has to gather information from two sensors, namely light and temperature. Each sensor and the network as a whole are allocated separate stacks that are tied up at the application level. Messages from each sensor are given a separate handler ID. When a network receives a message of a particular handler ID, it invokes the appropriate component. A typical message structure is shown in Table 1. Each packet coming out of a mote contains several fields of data. Table 1 gives a description of the various fields in the TOS_Msg data structure.

Byte #	Field	Description
0 - 1	Message Address	One of 3 possible value types: <ul style="list-style-type: none"> Broadcast Address (0xFFFF) – message to all nodes. UART Address (0x007e)– message from a node to the gateway serial port. All incoming messages will have this address. Node Address – the unique ID of a node to receive message.
2	Message Type	Active Message (AM) unique identifier for the type of message it is. Typically each application will have its own message type. Examples include: <ul style="list-style-type: none"> AMTYPE_XUART = 0x00 AMTYPE_MHOP_DEBUG = 0x03 AMTYPE_SURGE_MSG = 0x11 AMTYPE_XSENSOR = 0x32 AMTYPE_XMULTIHOP = 0x33 AMTYPE_MHOP_MSG = 0xFA
3	Group ID	Unique identified for the group of nodes participating in the network. The default value is 125 (0x7d). Only nodes with the same group ID will talk to each other.
4	Data Length	The length (<i>l</i>) in bytes of the data payload. This does not include the CRC or frame synch bytes.
5...n-2	Payload data	The actual message content. The data resides at byte 5 through byte 5 plus the length of the data (<i>l</i> from above). The data will be specific to the message type. Specific message types are discussed in the next section.
n-1, n	CRC	Two byte code that ensures the integrity of the message. The CRC includes the Packet Type plus the entire unescaped TinyOS message. A discussion on how the CRC is computed is included in the Appendix.

Table 1: Table showing various fields in a TOS_Msg data structure.

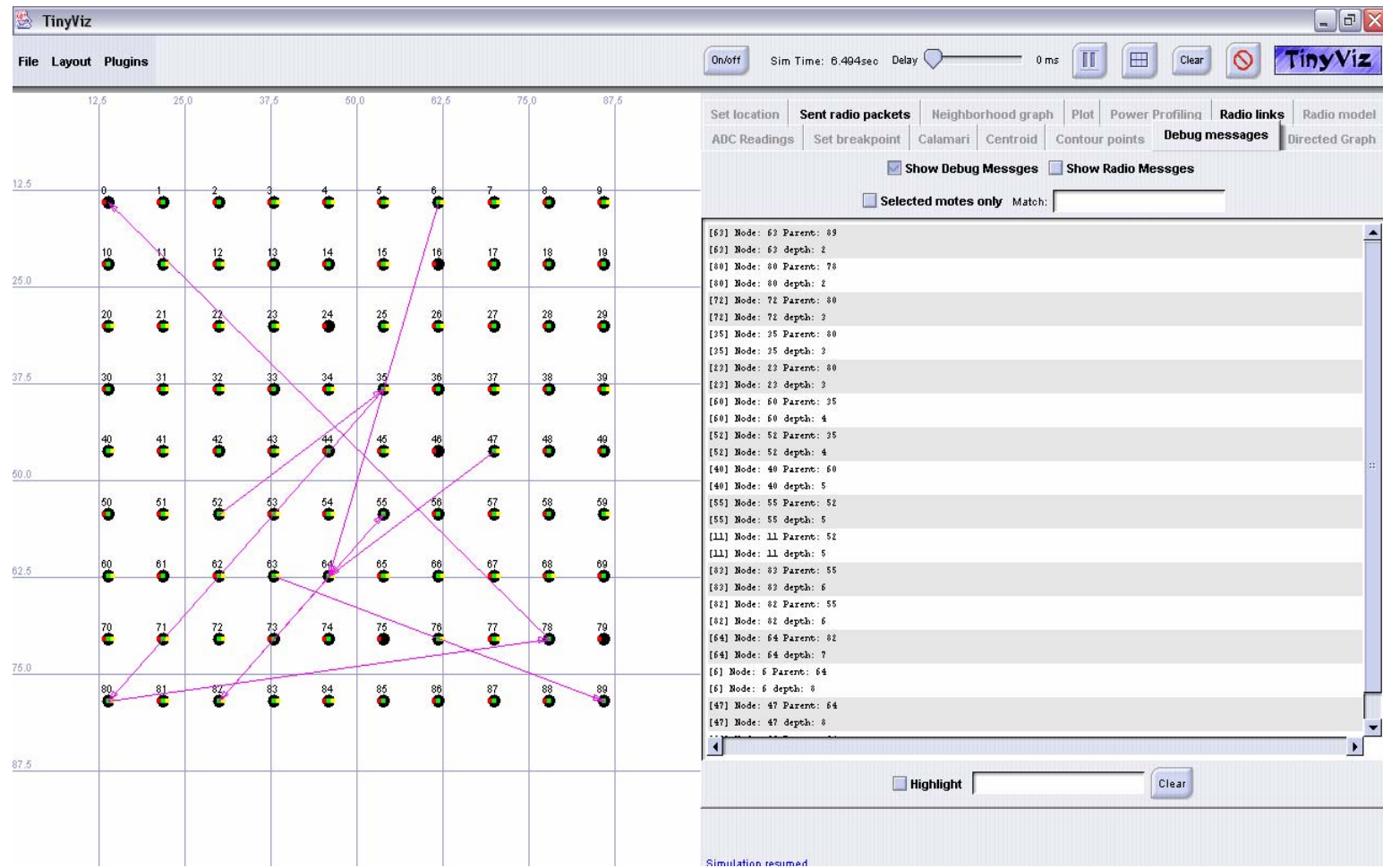


Fig. 2: A screenshot of the TinyViz software simulating the multi-hop protocol implemented on 90 sensor nodes.

3.3 TOSSIM

Apart from the different library components, Tiny OS also features a simulator that can be used to debug programs before they are downloaded into wireless sensor nodes. TOSSIM can be used to examine their code, and to test and analyze algorithms in a controlled and repeatable environment.

3.3.1 TinyViz

TinyViz is a Java visualization and actuation environment for TOSSIM. TinyViz uses a GUI to represent simulations that are run using TOSSIM. The proposed AMH algorithm was simulated using TOSSIM and visualized in TinyViz before it was implemented on WSN of T-Mote Sky nodes. A screenshot of the algorithm being simulated on a network of 90 nodes is shown in Fig. 2. Node zero represents the base station. It can be seen that all the nodes send their information back to the base station using multiple hops. The text on the right indicates the depth of each node in the network. An elaborate explanation of the algorithm and its results are presented in Chapter Four.

CHAPTER FOUR

AD-HOC AND MULTI-HOP ALGORITHM

The AMH algorithm enables the cooperating robots to communicate with each other as well as to communicate with the base station. The algorithm was coded using nesC language in a Tiny OS environment. The algorithm was uploaded from the base station to seven T-Motes that form the network using the USB port on the base station. The AMH algorithm not only provides a direct communication route between two nodes, it also enables multi-hop data routing between nodes that are on different tree depths. The algorithm also accounts for nodes that are added at a later stage or nodes that have become dysfunctional during the operation of the network.

This chapter is organized as follows. Initially, we describe the scenario of the implementation and then proceed to discuss the assumptions and the algorithm itself. Finally we conclude the chapter with snapshots and discussion of the results.

4.1 Scenario

The primary objective of the cooperative robotics research program at Auburn University is to build robots that can tackle real-world issues such as search and rescue missions, foraging (e.g. debris collection) or hazardous material cleanup using robots that

can “talk” to each other. The communication between the robots needs to adhere to certain protocols. As explained in chapter three, WSN’s serve as an excellent platform to perform our communication experiments. There are various wireless data routing protocols that currently exist; however there are no reliable or stable routing protocols for WSN’s. WSN’s are different from other wireless devices due to various reasons, but the primary reason is the severe memory constraint that makes it impossible to store routing/destination tables. WSN, though, are not the only solution for cooperative robotics research. Other platforms such as Gumstix/Robotstix [18] are also being experimented with. A final comparison between the various solutions needs to be done before deciding on a suitable communication platform for the cooperating robots.

The AMH algorithm was designed with WSN’s specifically in mind. The algorithm consists of three phases, which are described in detail in the following sections. To give an overview, the first phase is the initialization phase during which the base station sends out a beacon to initiate network formation. The second phase is the “parent node” allocation phase, during which each node identifies the node to which it will send its information. This phase is also used by nodes to identify their “depths” in the network tree. The third and final phase is the data collection and forwarding phase. In this phase, the nodes sample their ADCs and send information to one another as well as to the base station. This phase of the algorithm also deals with the ad-hoc concept of the network. There are provisions in the algorithm designed to govern the addition or deletion of a new node or an existing node respectively.

4.2 Assumptions

- 1) One of the nodes (node 0) is always connected to the base station.
- 2) The same program is running on all the existing nodes.
- 3) There are at least two levels of hierarchy in the network, i.e., there is at least one node in the network whose node depth is two.
- 4) All information collected by the nodes needs to be received at the base station.
- 5) When nodes are either added or deleted, there is always one node that is a single hop away from the base station.
- 6) The maximum information payload that each message can carry is 36 bytes.
- 7) During initial network formation each node is assigned a single, fixed parent. The parent may change during subsequent network operation.
- 8) Nodes with successive IDs are in direct communication range with each other.
- 9) Communicating nodes are at a maximum of 20 meters away from each other.
- 10) The base station is passive, i.e., it performs only data collection and does not issue any command during network operation. It is however responsible for initiating network formation by transmitting a beacon at the beginning.
- 11) There is no significant interference at 2.4 GHz from other RF devices.

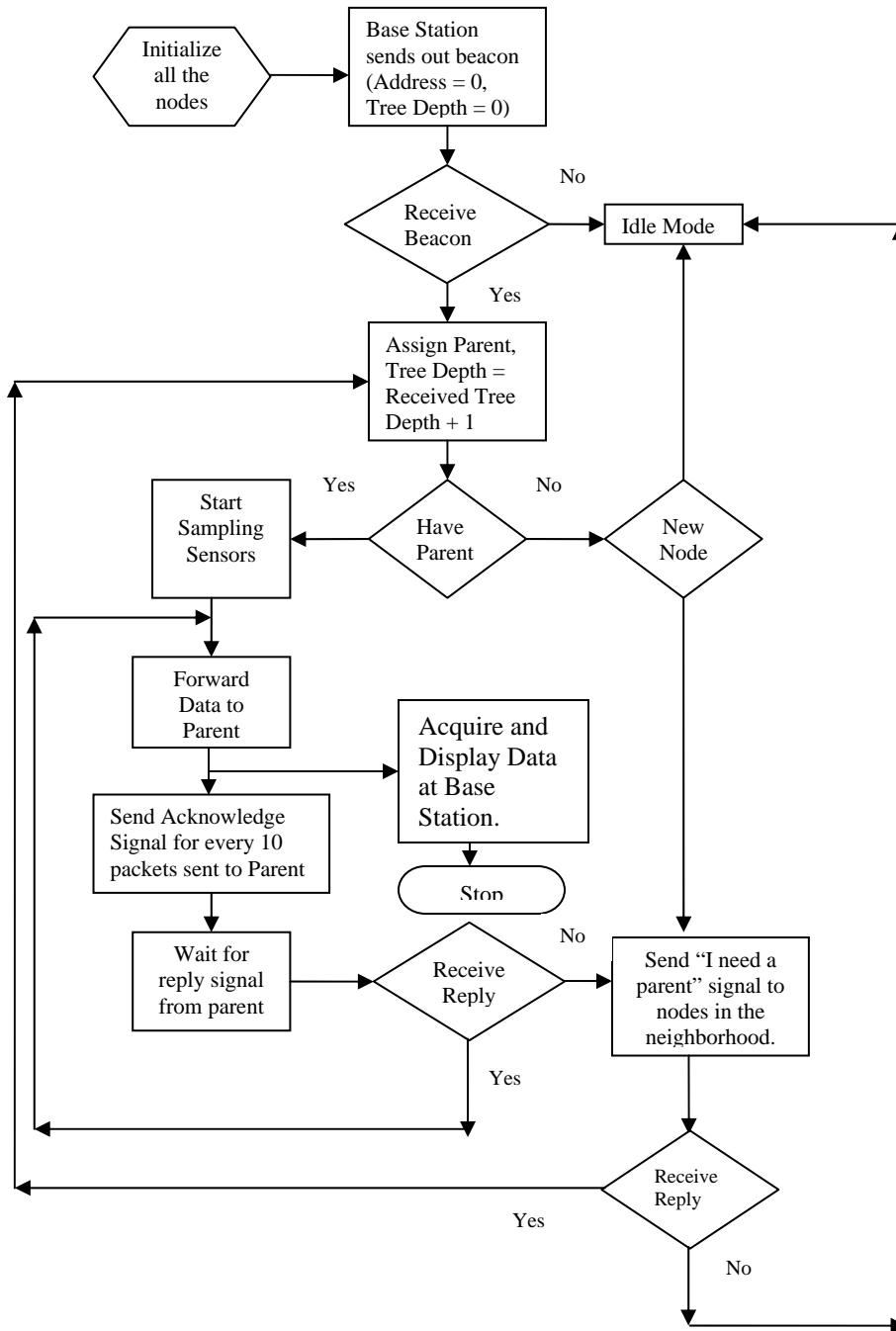


Fig. 3: Flowchart of the AMH algorithm.

4.3 Algorithm

The AMH algorithm is written using nesC. A flowchart describing the algorithm is shown in Fig. 3. The code implementing the algorithm is listed in Appendix A. The AMH algorithm essentially consists of three phases; (i) the initialization phase (ii) the network formation phase (iii) data collection and forwarding phase. Fig. 4 shows a diagrammatic representation of the experimental setup.

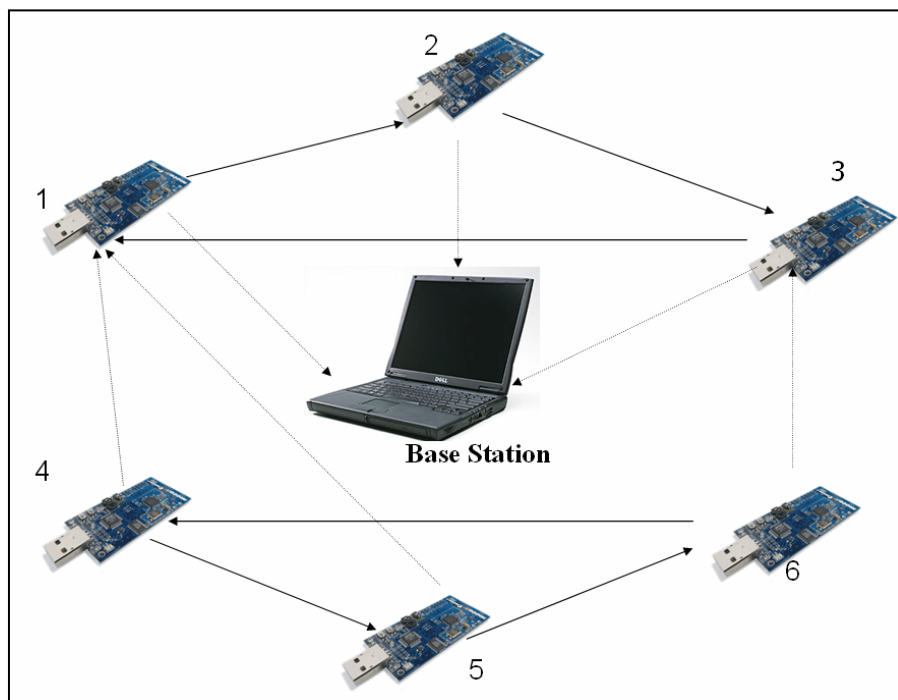


Fig. 4: Experimental Setup.

4.3.1 Initialization Phase

During this phase the nodes do not have an awareness of their depth in the network tree or the parent to which they should send data. The base station sends out a beacon advertising its node ID (zero) and tree depth (0). Nodes that can receive this

message set their parent to be zero and increment their node depth. A timer on the node at the base station is programmed to transmit this beacon for a total of five seconds in 100 ms intervals. Once the nodes receive the beacon, they re-transmit the beacon to other nodes that are out of direct communication range with the base station.

4.3.2 Network Formation Phase

Once the base station has sent the beacon, the first level of nodes identify the base station as their parent and retransmit the beacon to other nodes in the network. The transmitting nodes replace the existing node ID and tree depth with their own values. Receiving nodes set their parents to be the corresponding node from which they received the beacon. As soon as the nodes receive this beacon, they send a “Send Parent” signal requesting the beacon-transmitting node to be its parent. If the beacon-transmitting node does not have any other child, it sends an “Accept Parent” signal back to the child node accepting to be its parent and make appropriate adjustments by setting the “Have Child” flag. The “Have Child” flag is used to indicate whether a node has a child or not. This flag is used because of the following possible situation: when a node re-transmits a beacon, it can be heard by more than one node. All the nodes send a “Send Parent” signal back to the beacon-transmitting node. To avoid confusion at the beacon-transmitting node, the first node to respond becomes the child of the beacon-transmitting node. All other offers are rejected by the parent node.

Once each node has a parent, they also select a secondary parent. The secondary parent can be any node and is used as a backup when parent nodes fail during network operation.

4.3.3 Data Collection and Forwarding

Once the nodes assign themselves a parent, they also set a flag “Send Data” to indicate the same. As a next step, each node samples the multiple sensors available through the ADC onboard. The ADC sampling is an example particular to this case. Once a parent is assigned, the nodes are ready for communication and can transmit whatever information they like as long as it is made available to the radio in the format described in Chapter Three. The communication between the nodes is sent back to the base station either directly or through multiple hops. When a node sends a message to another node, a transcript is made that is sent to the node’s parent which in turn forwards it directly or sequentially to the base station. Whenever a node receives a data packet, it checks for two things: (i) whether it is destined destination for the packet and (ii) whether it has come from its own child or any other node. If the data packet does not satisfy both the conditions, it transfers the data packet to its parent and awaits the next message. A data packet is forwarded in this manner until it reaches the base station.

For every ten packets sent to the parent, each node sends an ACK (acknowledgement) signal. The ACK signal is used to check whether a parent node is still available or has become dysfunctional. If the parent node does not respond to the ACK signal within five seconds, the child node assigns the secondary parent as the

primary parent and proceeds to look for a new secondary parent. The interval of the ACK signal can be changed through software by altering relevant portions of the code in Appendix A.

When a new node is added, it issues an “I need a parent” signal. Nodes in the vicinity respond to this signal with their node ID’s and tree depths. The node with the least tree depth is selected as a parent and other nodes are rejected.

The following section presents various snapshots of the results (captured during the operation of the network) as well as a discussion of the user interface options made available to users who wish to study the data flow in the system.

4.4 Results and Discussions

This section presents the results of the experiments performed with the AMH algorithm. A program implementing the algorithm was written in nesC and downloaded onto seven T-Mote Sky modules. The experiments were done to demonstrate the ad-hoc and multi-hop features of the AMH algorithm. Following is a description of the experimental setup.

Initially three nodes are switched on. These three nodes form the first layer of communication modules that are in a direct communication range with the base station. Each of these nodes has a tree depth value of one. A wireless sensor node with ID zero is connected to a laptop through a USB port throughout the duration of the experiment. In this mode of operation, each node sends a message to another node in a cyclic manner. Node 1 sends a message to Node 2, Node 2 sends a message to Node 3 and Node 3

completes the chain by sending messages back to Node 1. It may be noted however that each node is capable of sending messages to any other node in a network. The arrangement described above is designed to reduce loss of messages due to interference. The second layer in the network tree consists of the next three nodes: Nodes 4, 5 and 6. The nodes in the second layer choose a parent from any of the nodes in the first layer. The algorithm is designed in such a way that each parent has a different child during the initial phases of the experiment. Later in the experiment, the parents may change due to various factors such as node failure, failure to acknowledge messages from child, etc. The nodes in the second layer have a tree depth value of two. Figure 6 is a screen shot of the initial arrangement of the experiment. Each data set has eight fields which are explained as follows:

Source: Source refers to the ID of the node. This ID is hardwired onto the mote during the downloading of the algorithm from the base station.

Parent: Parent is the ID of the node to which a source node forwards a transcript of its communication with any other node. Each node forwards its data transcript to its parent until it reaches the base station where it is read through node zero attached to its USB port.

Sec_Parent: Secondary Parent is a backup node incase the primary parent fails during network operation. As described in chapter four, each node sends an acknowledgement signal to its parent after sending ten data packets. If the parent fails to respond to the

acknowledgement message, the source node checks to see if the secondary parent node is available and if it is indeed available, changes its parent ID to the secondary parent node ID. This is done to reduce the time taken by nodes searching for a new parent to send their data. If time is not a constraint, this section of code can be done away with to save significant program memory space on the nodes.

Tree Depth (Node Depth): This is indicative of the position of each node in the network tree.

ID: ID is a measure of the number of messages sent by each node. A message ID is increased only if it originates from a node. It cannot be incremented by the parent or secondary parent nodes.

Sent Destination: Sent destination refers to the destination of each message originating from a source node.

Data: This field can store up to 21 bytes of data to be communicated between various nodes. In the present case, each node samples its light sensor and shares the reading with other nodes. Since the experiment was carried out in an indoor environment with constant light intensity, all the nodes exchange a value of 0x07.

Timex: This field has a provision to store two bytes of time data. In the experiments carried out, this field was not used due to the absence of time synchronization feature in

the algorithm. Once this is implemented, the timex field can be used to carry the time when each message was sent from a particular node.

To demonstrate the ad-hoc nature of the algorithm, we disrupt the functioning of the network by forcing a node to breakdown. It will be shown that the network can reconfigure itself to compensate the loss of a node. In Fig. 5 and Fig. 6 we observe that Node 6 has assigned its parent to be node three. We force Node 3 to fail and observe in Figures 7 - 11 how Node 6 identifies another parent, Node 5.

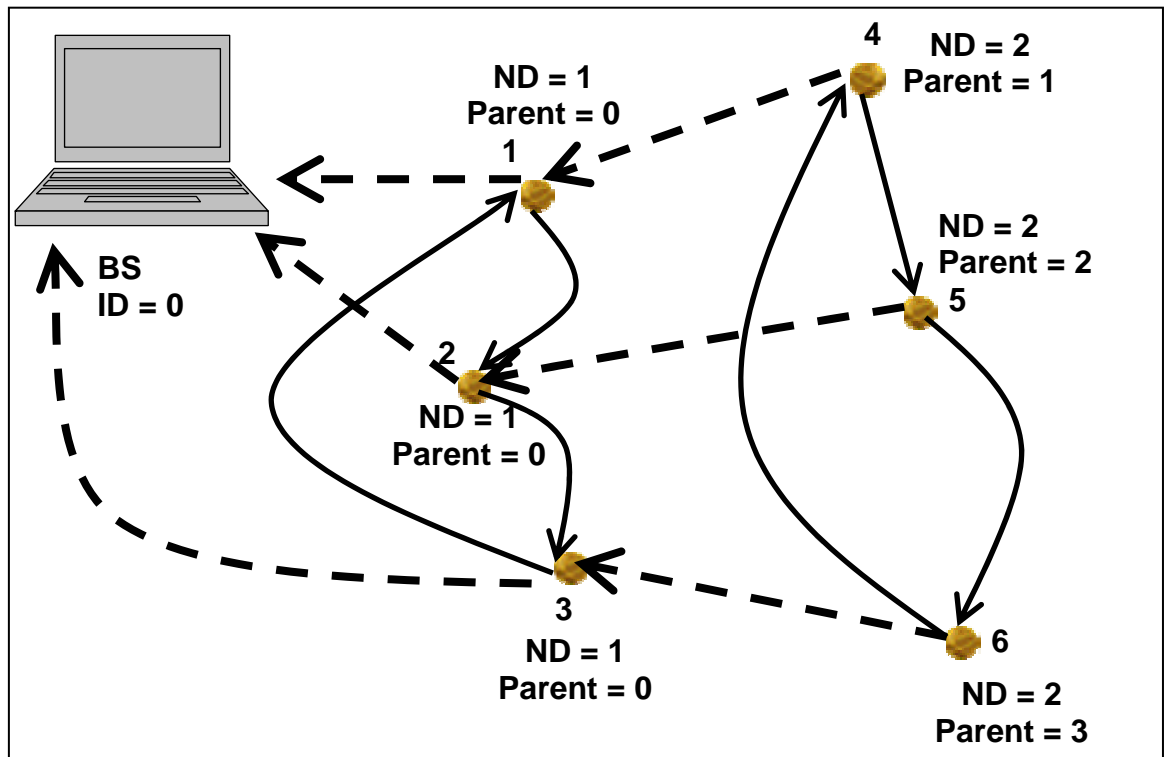


Fig. 5: A graphical representation of the initial setup of the network.

```
[timex=0x61 ]
TimeMsg:Message <TimeMsg>
[source=0x1 ]
[parent=0x0 ]
[sec_parent=0x0 ]
[tree_depth=0x1 ]
[id=0x9 ]
[sent_destination=0x2 ]
[data=0x7 ]
[timex=0xcf ]

TimeMsg:Message <TimeMsg>
[source=0x5 ]
[parent=0x2 ]
[sec_parent=0x1 ]
[tree_depth=0x2 ]
[id=0x2 ]
[sent_destination=0x6 ]
[data=0x7 ]
[timex=0x9d ]

TimeMsg:Message <TimeMsg>
[source=0x2 ]
[parent=0x0 ]
[sec_parent=0x0 ]
[tree_depth=0x1 ]
[id=0x9 ]
[sent_destination=0x3 ]
[data=0x7 ]
[timex=0xa7 ]

TimeMsg:Message <TimeMsg>
[source=0x6 ]
[parent=0x3 ]
[sec_parent=0x3 ]
[tree_depth=0x2 ]
[id=0x1 ]
[sent_destination=0x4 ]
[data=0x7 ]
[timex=0xcf ]

TimeMsg:Message <TimeMsg>
[source=0x3 ]
[parent=0x0 ]
[sec_parent=0x0 ]
[tree_depth=0x1 ]
[id=0x9 ]
[sent_destination=0x1 ]
[data=0x7 ]
[timex=0x4d ]

TimeMsg:Message <TimeMsg>
[source=0x4 ]
[parent=0x1 ]
[sec_parent=0x1 ]
[tree_depth=0x2 ]
[id=0x2 ]
[sent_destination=0x5 ]
[data=0x7 ]
[timex=0xa7 ]
```

Fig. 6: A screenshot of the readings at the base station during the initial phase of the network.

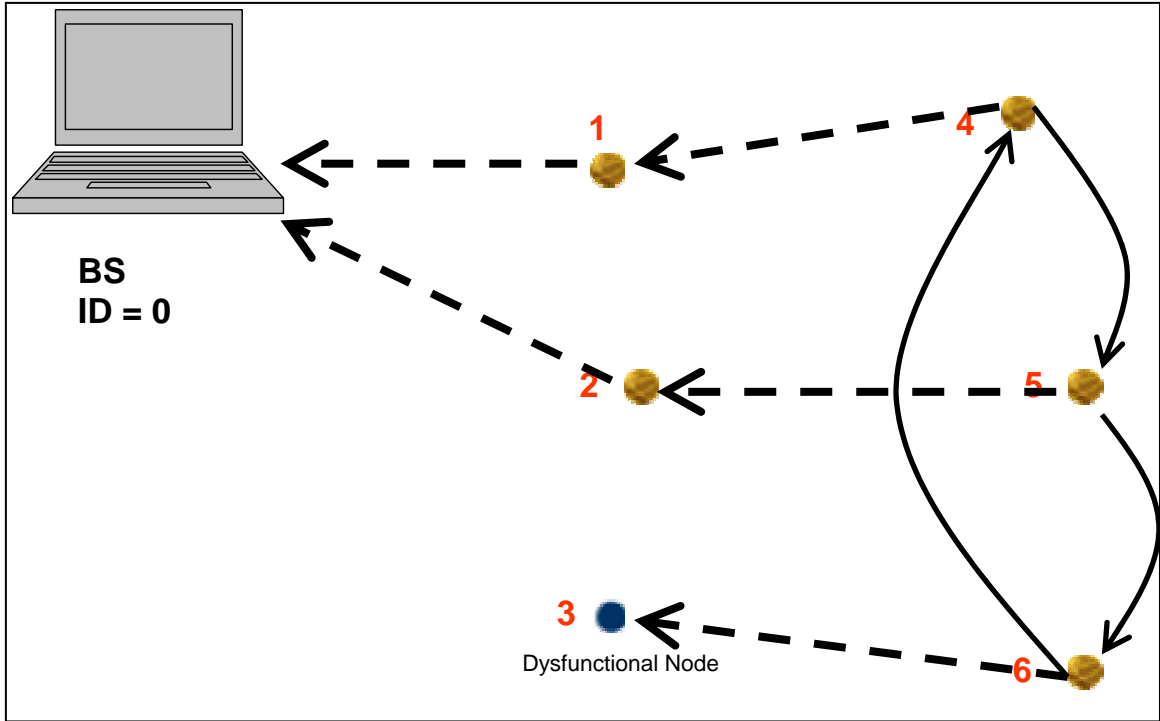


Fig. 7: A graphical representation of Node 3 breaking down.

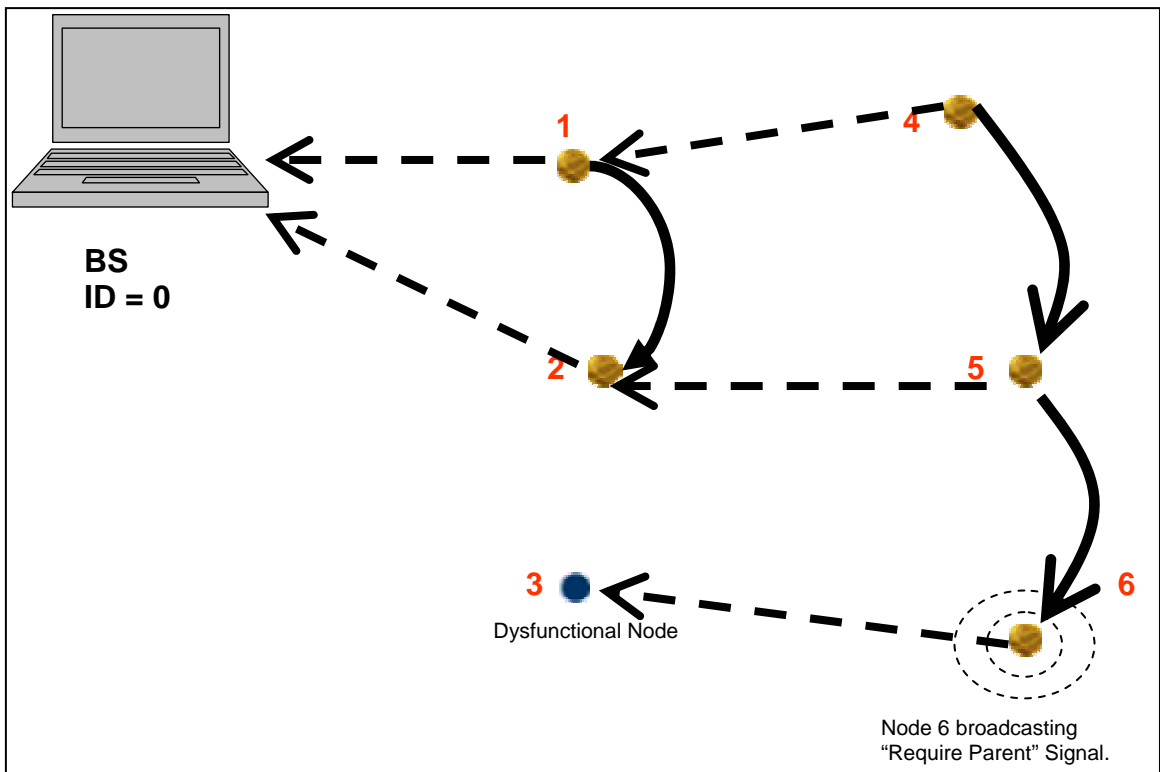


Fig. 8: A graphical representation of Node 6 requesting a new parent.

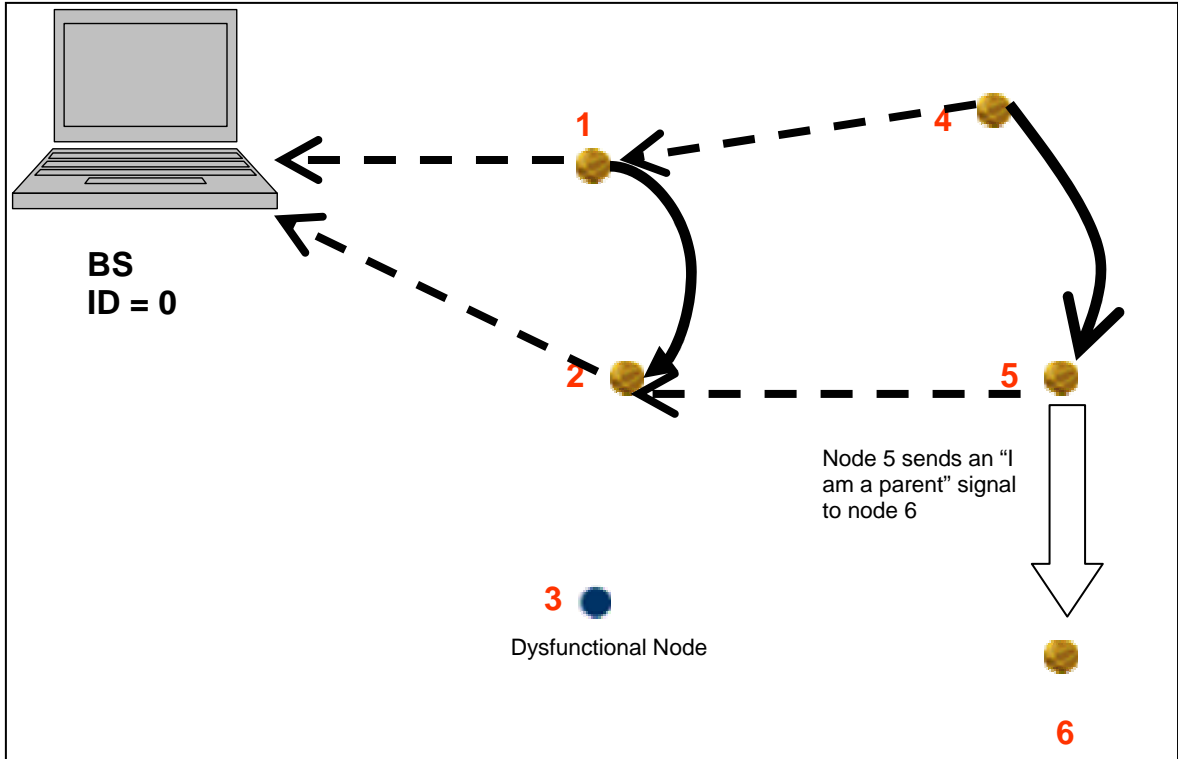


Fig. 9: A graphical representation of Node 5 responding to Node 6 with an "I am a Parent" signal.

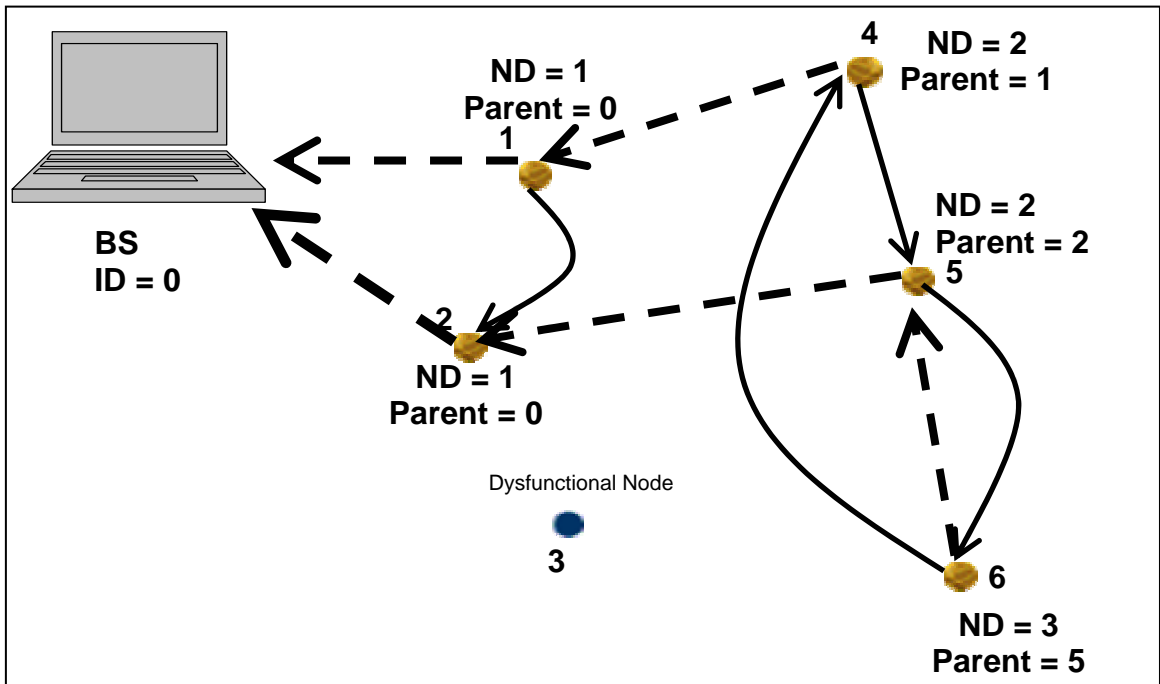


Fig. 10: A graphical representation of the changed parent of Node 6.

```
TimeMsg:Message <TimeMsg>
[source=0x1]
[parent=0x0]
[sec_parent=0x0]
[tree_depth=0x1]
[id=0x22]
[sent_destination=0x2]
[data=0x7]
[time_x=0xf184]

TimeMsg:Message <TimeMsg>
[source=0x4]
[parent=0x1]
[sec_parent=0x1]
[tree_depth=0x2]
[id=0x3c]
[sent_destination=0x5]
[data=0x7]
[time_x=0x278]

TimeMsg:Message <TimeMsg>
[source=0x4]
[parent=0x1]
[sec_parent=0x6]
[tree_depth=0x2]
[id=0x3d]
[sent_destination=0x5]
[data=0x7]
[time_x=0x808]

TimeMsg:Message <TimeMsg>
[source=0x5]
[parent=0x2]
[sec_parent=0x4]
[tree_depth=0x2]
[id=0x45]
[sent_destination=0x6]
[data=0x7]
[time_x=0x300]

TimeMsg:Message <TimeMsg>
[source=0x2]
[parent=0x0]
[sec_parent=0x0]
[tree_depth=0x1]
[id=0x25]
[sent_destination=0x3]
[data=0x7]
[time_x=0x1d]

TimeMsg:Message <TimeMsg>
[source=0x6]
[parent=0x5]
[sec_parent=0x5]
[tree_depth=0x2]
[id=0x3c]
[sent_destination=0x4]
[data=0x7]
[time_x=0x62]
```

Fig. 11: Screenshot displaying the changed parent of node six. The parent was changed from Node 3 to Node 5.

A demonstration of the ad-hoc nature of the algorithm is not complete without proving its ability to add new nodes to the network. To simulate this behavior, we bring Node 3 back into the network. Figures 12 -15 show how Node 3 fits seamlessly back into the network. As an additional option, the parent that Node 3 chooses also transmits the latest ID that it has in its memory to enable continuity in the network. For example, when Node 3 re-enters the network, it sends out a signal (“I need a parent” signal) requesting a parent. Nodes in the vicinity check their memories to verify if they have received any messages from Node 3 before it broke down. If they have received a message from Node 3, they send the corresponding ID along with an acknowledgement to the “I need a parent signal”. Node 3 examines the received information from each node and chooses as its parent, the node which sent a message with the most recent ID. In a real world application, this can help replacement robots continue from the point where their former counterparts broke down. In case the node is new, it chooses as a parent, the first node from which it receives an acknowledgement for the “I need a parent” signal.

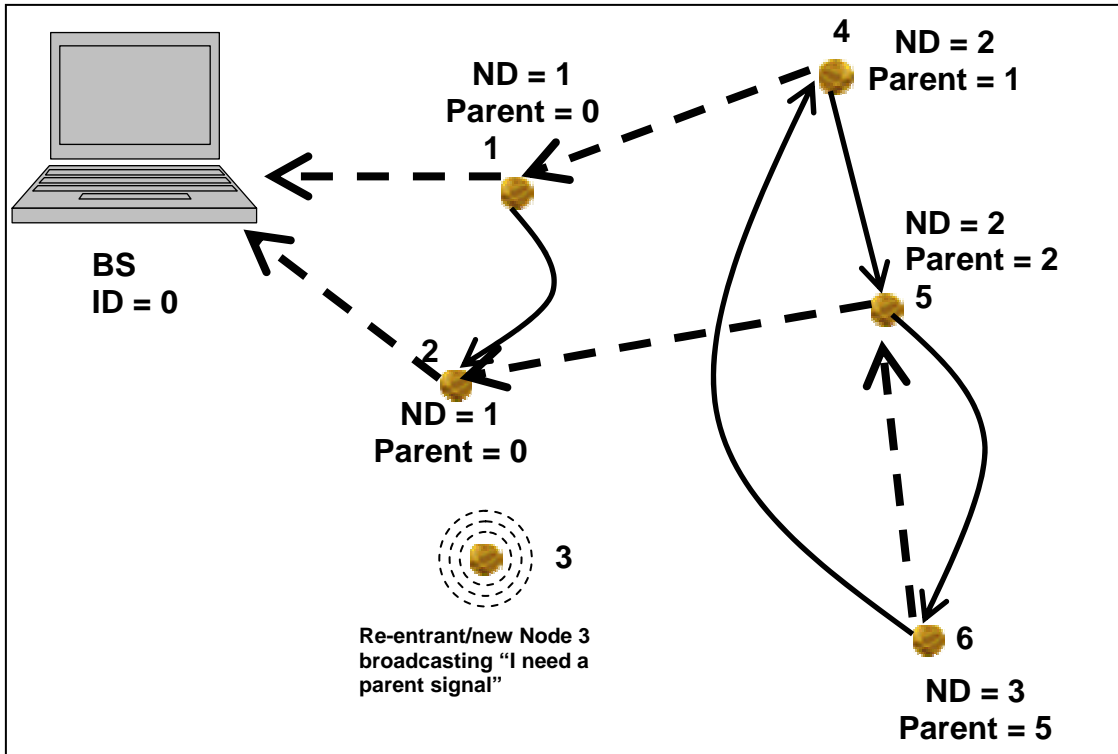


Fig. 12: A graphical representation of Node 3 re-entering the network.

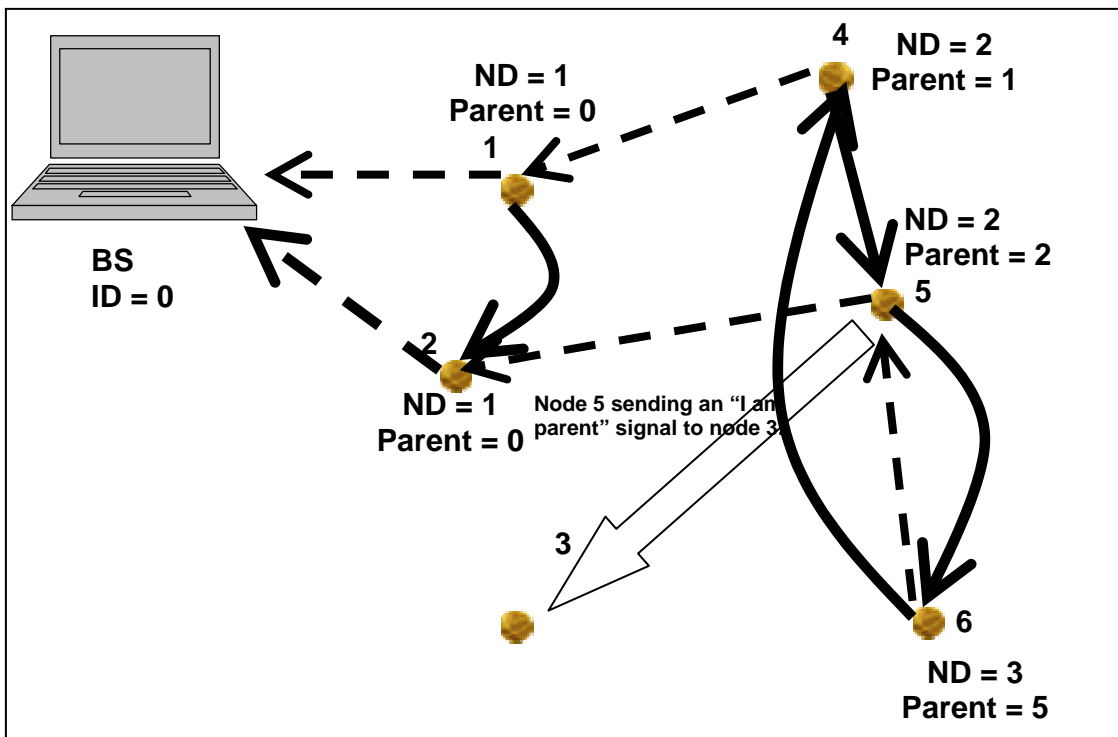


Fig. 13: A graphical representation of Node 5 responding to Node 3's request.

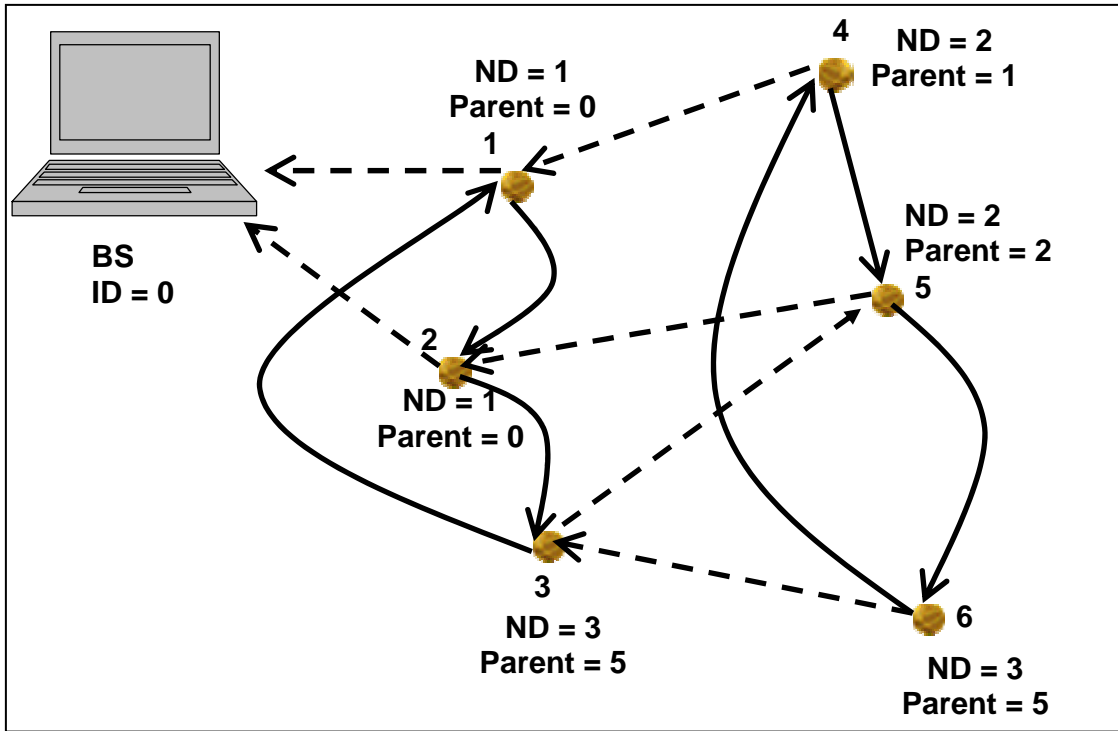


Fig. 14: A graphical representation of the seamless re-entry of Node 3 into the network.

```
[timex=0xa7]
TimeMsg:Message <TimeMsg>
[source=0x3]
[parent=0x5]
[sec_parent=0x4]
[tree_depth=0x3]
[id=0x21]
[sent_destination=0x1]
[data=0x7]
[timex=0x2c8]

TimeMsg:Message <TimeMsg>
[source=0x5]
[parent=0x1]
[sec_parent=0x6]
[tree_depth=0x2]
[id=0x76]
[sent_destination=0x6]
[data=0x7]
[timex=0x264]

TimeMsg:Message <TimeMsg>
[source=0x5]
[parent=0x1]
[sec_parent=0x4]
[tree_depth=0x2]
[id=0x77]
[sent_destination=0x6]
[data=0x7]
[timex=0x822]

TimeMsg:Message <TimeMsg>
[source=0x1]
[parent=0x0]
[sec_parent=0x0]
[tree_depth=0x1]
[id=0x43]
[sent_destination=0x2]
[data=0x7]
[timex=0xa9]

TimeMsg:Message <TimeMsg>
[source=0x4]
[parent=0x1]
[sec_parent=0x1]
[tree_depth=0x2]
[id=0x72]
[sent_destination=0x5]
[data=0x7]
[timex=0x221]

TimeMsg:Message <TimeMsg>
[source=0x5]
[parent=0x4]
[sec_parent=0x3]
[tree_depth=0x2]
[id=0x78]
[sent_destination=0x6]
[data=0x7]
[timex=0x308]
```

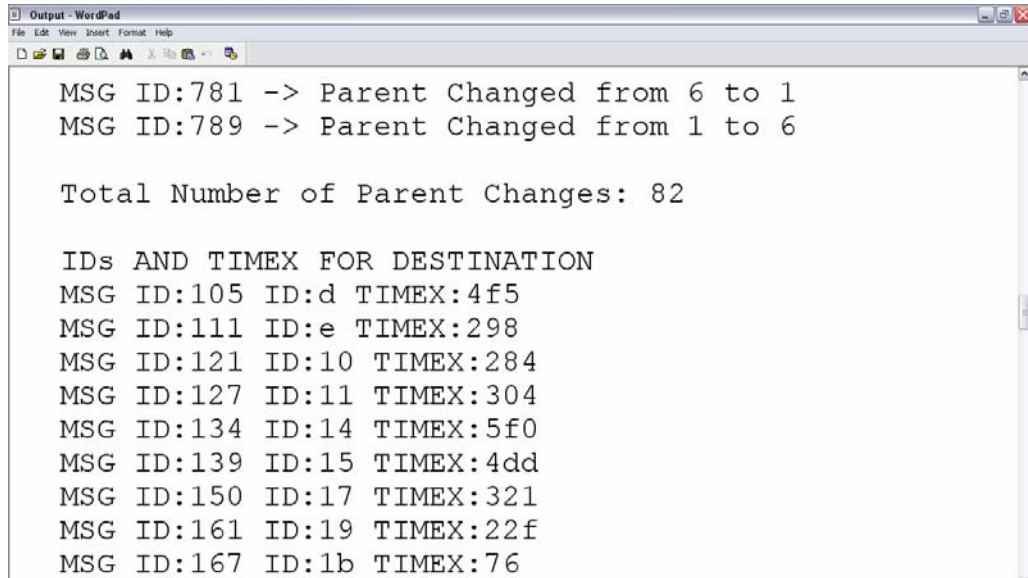
Fig. 15: Screen shot displaying the seamless reentry of Node 3 into the network.

4.4.1 User Interface Discussion

The message transcripts sent from the various nodes is displayed on the base station using a java program that is described in APPENDIX B. The incoming data is both displayed on the screen as well as parsed into a text file. The text file was created to record the events happening during the operation of the network. A screenshot of the text file is shown in Fig. 16. The text file displays the data fields present in each message. A user interface was programmed using perl. After studying the text file, the perl script is run and the values of the source node and message ID are entered. The user interface then generates a text file that lists the source node and destination node of the particular message ID, the number of times the parent of the source node has changed and the time at which the change took place. The text file further displays the ID of the messages that were sent out of the destination node in response to the message sent by the source node. The idea of the user interface is to track the events happening in the network with the help of the message IDs. The use of the interface is explained with the following example. Let us assume that we wish to see the effect of Node 4 sending a message to Node 5 at a given time (timex field). The message ID corresponding to that time and source node is entered in the user interface. The script then parses the text file to generate the messages that were sent from Node 5 after the given time. The output file also gives details about the parent changes of Node 4 and the time of occurrence of the changes. A screenshot of the generated output file is shown in Fig. 17.


```
Output - WordPad
File Edit View Insert Format Help
RESULTS LOG
Date: 7-15-2006
Input File: C:\Program Files\UCB\cygwin\opt\tinyos-1.x\tools\java\net\tinyos\Ad_hoc\log.txt
*****
*****
SOURCE      : 4
DESTINATION : 5
CHANGE IN PARENTS
MSG ID:107 -> Parent Changed from 5 to 3
MSG ID:117 -> Parent Changed from 3 to 6
MSG ID:123 -> Parent Changed from 6 to 3
MSG ID:129 -> Parent Changed from 3 to 6
MSG ID:135 -> Parent Changed from 6 to 2
MSG ID:146 -> Parent Changed from 2 to 1
MSG ID:153 -> Parent Changed from 1 to 6
MSG ID:163 -> Parent Changed from 6 to 2
MSG ID:176 -> Parent Changed from 2 to 1
MSG ID:183 -> Parent Changed from 1 to 2
MSG ID:189 -> Parent Changed from 2 to 1
MSG ID:195 -> Parent Changed from 1 to 2
MSG ID:204 -> Parent Changed from 2 to 3
MSG ID:210 -> Parent Changed from 3 to 1
MSG ID:230 -> Parent Changed from 1 to 2
MSG ID:237 -> Parent Changed from 2 to 6
MSG ID:249 -> Parent Changed from 6 to 1
```

Fig. 17: A screenshot showing the upper half of the output file.



```
Output - WordPad
File Edit View Insert Format Help
MSG ID:781 -> Parent Changed from 6 to 1
MSG ID:789 -> Parent Changed from 1 to 6

Total Number of Parent Changes: 82

IDs AND TIMEX FOR DESTINATION
MSG ID:105 ID:d TIMEX:4f5
MSG ID:111 ID:e TIMEX:298
MSG ID:121 ID:10 TIMEX:284
MSG ID:127 ID:11 TIMEX:304
MSG ID:134 ID:14 TIMEX:5f0
MSG ID:139 ID:15 TIMEX:4dd
MSG ID:150 ID:17 TIMEX:321
MSG ID:161 ID:19 TIMEX:22f
MSG ID:167 ID:1b TIMEX:76
```

Fig. 18: A screenshot showing the bottom part of the output file.

CHAPTER FIVE

CONCLUSIONS

An ad-hoc, multi-hop algorithm (AMH) for WSN used in cooperative robotics has been presented. The AMH algorithm has been shown to work by implementing a self-forming network using seven wireless sensor nodes. The algorithm is highly flexible and scalable to accommodate more than seven nodes in the network. The sensor nodes can work both in indoor and outdoor environments although their communication range varies depending upon where they are deployed. The algorithm is flexible to accommodate any variation during network operation due to a node failure or node addition.

Experiments were conducted in an indoor environment and three features of the AMH algorithm were monitored: the ability of individual nodes to communicate with each other, the ability of the network to react to any changes such as node failures or node accessions and finally the ability to send transcripts of exchanged messages back to the base station.

According to the author's knowledge, this thesis has presented one of the first algorithms for WSN in cooperative robotics. Although WSN have been used in cooperative robotics, there have been few instances where they have been used for inter-robot communication as well as base station monitoring.

This thesis has provided a valuable insight into the field of WSN for inter-robotic communications and a significant foundation for the CRR group at Auburn University. At present there are robots being built, that are capable of using WSN nodes such as T-Mote Sky, as their communication module. This thesis should hopefully inspire fellow CRR team members to have an in-depth look at the WSN domain and tap its ever-increasing applications in other aspects of cooperative robotics such as localization and time synchronization.

5.1 Suggestions for Future Work

- To have hardware demonstration of the robotic network with a group of 5 to 10 robots.
- To do an analysis of system overhead related to event logging.
- Perform a qualitative analysis of the AMH algorithm to optimize communication parameters thereby maximizing global network resource utilization.
- Replace WSN with Gumstix/Robostix as communication module for cooperative robotics. Develop a version of the AMH algorithm for the Gumstix/Robostix platform and perform a comparison between the two platforms (WSN and Robostix) to determine the ideal communication module for cooperative robotics.

REFERENCES

- [1] L. Almeida, J. L. Azevedo, P. Bartolomeu, E. Brito, M. B. Cunha, J. P. Figueiredo, P. Fonseca, C. Lima, R. Marau, N. Lau, P. Pedreiras, A. Pereira, A. Pinho, F. Santos, L. S. Lopes, and J. Vieira, "CAMBABDA:Team Description Paper," *RoboCup Symposium: Papers and Team Discussion Papers*, 2004.
- [2] H. Asama, A. Matsumoto, and Y. Ishida, "Design of an autonomous and distributed robot system: ACTRESS," *IEEE/RSJ IROS*, pp. 283-290, 1989.
- [3] T. Balch and R. C. Arkin, "Communication in Reactive Multiagent Robotic Systems," *Autonomous Robots*, vol. 1, pp. 27-52, 1994.
- [4] D. Barnes and J. Gray, "Behavior synthesis for cooperant mobile robot control.," *International Conference on Control*, pp. 1135-1140, 1991.
- [5] M. A. Batalin, G. S. Shukhatme, and M. Hattig, "Mobile robot navigation using a sensor network," presented at IEEE International Conference on Robotics & Automation, 2003.
- [6] P. Bauer, M. Sichitiu, R. Istebanian, and K. Premaratne, "The mobile patient: wireless distributed sensor networks for patient monitoring and care," *International Conference on Information Technology Applications in Biomedicine*, pp. 17-21, 2000.
- [7] G. Beni, "The concept of cellular robotic system," *IEEE International Symposium on Intelligent Control*, pp. 57-62, 1988.
- [8] V. P. Burhanpurkar, "Real world application of a low-cost high-performance sensor system for autonomous mobile robots," presented at Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems, Munich, Germany, 1994.
- [9] Y. U. Cao, A. S. Fukunaga, and A. B. Kahng, "Cooperative mobile robotics: Antecedents and directions," *Autonomous Robots*, vol. 4, pp. 7-27, 1997.
- [10] M. Carlson, C. Christiansen, A. Persson, E. Jonsson, M. Seeman, and H. Vindahl, "RoboCupRescue - Robot League Team," presented at RoboCup 2004, Lisbon, Portugal, 2004.

- [11] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao, "Habitat monitoring: Application driver for wireless communications technology," *Computer Communication Review*, vol. 31, pp. 20+, 2001.
- [12] L. Chaimowicz, T. Sugar, V. Kumar, and M. F. M. Campos, "An Architecture for Tightly Coupled Multi-Robot Cooperation," presented at IEEE International Conference on Robotics and Automation, Seoul, Korea, 2001.
- [13] A. Das, G. Kantor, V. Kumar, G. Pereira, R. Peterson, D. Rus, S. Singh, and J. Spletzer, "Distributed Search and Rescue with Robot and Sensor Teams," presented at International Conference on Field and Service Robotics, 2003.
- [14] R. Dorf, *Concise International Encyclopedia of Robotics: Applications and Automation*: Wiley-Interscience, 1990.
- [15] J. Elson and D. Estrin, "Time Synchronization for Wireless Sensor Networks," presented at Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing, San Francisco, CA, USA, 2001.
- [16] T. Fukuda and S. Nakagawa, "A dynamically reconfigurable robotic system (concept of a system and optimal configurations)," *International Conference on Industrial Electronics, Control and Instrumentation*, pp. 588-595, 1987.
- [17] B. P. Gerkey, R. T. Vaughn, K. Stoy, A. Howard, G. S. Sukhatme, and M. J. Mataric, "Most Valuable Player: A Robot Device Server for Distributed Control," *IEEE/RSJ IROS*, pp. 1226-1231, 2001.
- [18] Gumstix/Robostix Home Page: <http://www.gumstix.com/>
- [19] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors," *Architectural Support for Programming Languages and Operating Systems*, pp. 93--104, 2000.
- [20] A. Howard, L. E. Parker, and G. S. Sukhatme, "Experiments with a large heterogeneous mobile robot team: Exploration, mapping, deployment and detection," *International Journal of Robotics Research*, vol. 25, pp. 431-447, 2006.
- [21] N. Hutin, C. Pegard, and E. Brassart, "A Communication Strategy for cooperative robots," presented at International Conference on Intelligent Robots and Systems, Victoria, B.C., Canada, 1998.
- [22] M. Matriac, "Interaction and Intelligent Behavior.," in *EECS*: MIT, 1994.

- [23] M. J. Matriac, "Behavior-based systems: Key properties and implications," presented at IEEE International Conference on Robot Automation, Nice, France, 1992.
- [24] Mica2 Homepage: <http://www.xbow.com/mica2>.
- [25] MicaZ. Homepage: <http://www.xbow.com/micaz>.
- [26] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," presented at PLDI, San Diego, 2003.
- [27] L. E. Parker, "Current State of the Art in Autonomous Distributed Mobile Robotics," *Distributed Autonomous Robotic Systems*, vol. 4, pp. 3-12, 2000.
- [28] R. Peterson and D. Rus, "Interacting with a Sensor Network," presented at 2002 Australian Conference on Robotics and Automation, Auckland, NZ, 2002.
- [29] S. Premvuti and S. Yuta, "Consideration on the cooperation of multiple autonomous mobile robots," *IEEE/RSJ IROS*, pp. 59-63, 1990.
- [30] A. Ray, "Cooperative Robotics using Wireless Communication," in *Electrical and Computer Engineering*. Auburn: Auburn University, 2005, pp. 183.
- [31] E. M. Royer and C.-K. Toh, "A review of current routing protocols for ad hoc mobile wireless networks," *IEEE Personal Communications*, vol. 6, pp. 46-55, 1999.
- [32] D. Rus and K. Kotay, "Versatility for unknown worlds: Mobile sensors and self-reconfiguring robots.," *Field and Service Robotics*, 1998.
- [33] M. Schneider-Fontan and M. J. Matriac, "Territorial Multi-Robot Task Division," *IEEE Transactions on Robotics and Automation*, vol. 14, pp. 815-822, 1998.
- [34] R. Simmons, D. Apfelbaum, W. Burgard, D. Fox, M. Moors, S. Thrun, and H. Younes, "Coordination for Multi-Robot Exploration and Mapping," presented at National Conference on Artificial Intelligence, 2000.
- [35] A. Stroupe, M. C. Martin, and T. Balch, "Distributed Sensor Fusion for Object Position Estimation by Multi-Robot Systems," presented at IEEE International Conference on Robotics and Automation, May, 2001.
- [36] Tiny OS Homepage: <http://www.tinyos.net>.

- [37] D. J. Turnell and M. D. F. Q. V. Turnell, "SimBot - A simulation tool for autonomous robots," presented at IEEE International Conference on Systems, Manufacturing and Cybernetics., 2001.
- [38] O. Wijk, P. Jensfelt, and B. Wahlberg, "Sensor Fusion for Mobile Robot Navigation - A First Subjective Discussion," 1997.
- [39] F. Zhao and L. Guibas, *Wireless Sensor Networks: An Information Processing Approach*: Morgan Kaufmann, 2005.

APPENDICES

APPENDIX A

CODE IMPLEMENTATION OF AMH ALGORITHM

This appendix presents the code for implementation of the AMH algorithm. This code is written in nesC and consists of two parts: the configuration file and the module file. As mentioned before Tiny OS has several abstracted components and the configuration file is responsible for mapping and bringing together required components. The module file is responsible for the actual implementation of the algorithm. The header files, configuration and module files must all be present in the same folder for proper execution.

Header Files:

Time:

```
typedef struct TimeMsg {
    uint16_t source;
    uint16_t parent;
    uint16_t sec_parent;
    uint16_t tree_depth;
    uint16_t id;
    uint16_t sent_destination;
    uint16_t data;
    uint16_t timex;
} TimeMsg;

enum {
    AM_TIMEMSG = 42
};
```

Miscellaneous:

```
typedef struct DataMsg {  
    uint16_t source;  
    uint16_t destinat;  
    uint16_t data;  
} DataMsg;
```

```
typedef struct Beacon {  
    uint16_t sourceaddress;  
    uint16_t dest;  
    uint16_t tree_depth;  
} Beacon;
```

```
typedef struct NewMsg {  
    uint16_t source;  
} NewMsg;
```

```
typedef struct AckMsg {  
    uint16_t source;  
} AckMsg;
```

```
typedef struct ReplyAckMsg {  
    uint16_t source;  
} ReplyAckMsg;
```

```
typedef struct ParentMsg {  
    uint16_t source;  
    uint16_t tree_depth;  
} ParentMsg;
```

```
typedef struct SecParentMsg {  
    uint16_t source;  
    uint16_t tree_depth;  
} SecParentMsg;
```

```
typedef struct SecparentMsg {  
    uint16_t source;  
    uint16_t tree_depth;  
} SecparentMsg;
```

```
typedef struct AcceptMsg {  
    uint16_t source;  
} AcceptMsg;
```



```

typedef struct AcceptSecMsg {
    uint16_t source;
} AcceptSecMsg;

enum {
    AM_DATA = 41,
    AM_BEACON = 43,
    AM_NEWMSG = 44,
    AM_PARENTMSG = 45,
    AM_SECMSG = 46,
    AM_ACCEPTMSG = 47,
    AM_SECONDARYMSG = 48,
    AM_ASMSG = 49,
    AM_ACKMSG = 50,
    AM_REPLYMSG = 51
};

```

Configuration File:

```

includes Miscellaneous;
includes Time;

```

```

configuration Coop{
}
implementation {
    components Main, CoopM, TimerC, LedsC, DemoSensorC as Sensor,
    GenericComm as Comm;

```

```

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

```

```

CoopM.RouteTimer -> TimerC.Timer[unique("Timer")];
CoopM.DelayTimer -> TimerC.Timer[unique("Timer")];
CoopM.AckTimer -> TimerC.Timer[unique("Timer")];

```

```

CoopM.ADC -> Sensor;
CoopM.ADCControl -> Sensor;

```

```

CoopM.Leds -> LedsC;

```

```

CoopM.SendBeacon -> Comm.SendMsg[AM_BEACON];
CoopM.ReceiveBeacon -> Comm.ReceiveMsg[AM_BEACON];
CoopM.SendDataPacket -> Comm.SendMsg[AM_DATA];
CoopM.ReceiveDataPacket -> Comm.ReceiveMsg[AM_DATA];

```

```

CoopM.SendTime -> Comm.SendMsg[AM_TIMEMSG];
CoopM.ReceiveTime -> Comm.ReceiveMsg[AM_TIMEMSG];
CoopM.SendIamNew -> Comm.SendMsg[AM_NEWMSG];
CoopM.ReceiveIamNew -> Comm.ReceiveMsg[AM_NEWMSG];
CoopM.SendParent -> Comm.SendMsg[AM_PARENTMSG];
CoopM.ReceiveParent -> Comm.ReceiveMsg[AM_PARENTMSG];
CoopM.SendSecParent -> Comm.SendMsg[AM_SECMSG];
CoopM.ReceiveSecParent -> Comm.ReceiveMsg[AM_SECMSG];
CoopM.SendAcceptParent -> Comm.SendMsg[AM_ACCEPTMSG];
CoopM.ReceiveAcceptParent -> Comm.ReceiveMsg[AM_ACCEPTMSG];
CoopM.SendSecondaryParent -> Comm.SendMsg[AM_SECONDARYMSG];
CoopM.ReceiveSecondaryParent -> Comm.ReceiveMsg[AM_SECONDARYMSG];
CoopM.SendAcceptSecParent -> Comm.SendMsg[AM_ASMSG];
CoopM.ReceiveAcceptSecParent -> Comm.ReceiveMsg[AM_ASMSG];
CoopM.SendAck -> Comm.SendMsg[AM_ACKMSG];
CoopM.ReceiveAck -> Comm.ReceiveMsg[AM_ACKMSG];
CoopM.SendReplyAck -> Comm.SendMsg[AM_REPLYMSG];
CoopM.ReceiveReplyAck -> Comm.ReceiveMsg[AM_REPLYMSG];
}

```

Module File:

```

includes Miscellaneous;
includes Time;

module CoopM {
    provides {
        interface StdControl;
    }
    uses {
        interface Timer as RouteTimer;
        interface Timer as DelayTimer;
        interface Timer as AckTimer;
        interface Leds;
        interface ADC;
        interface StdControl as ADCControl;

        interface SendMsg as SendBeacon;
        interface ReceiveMsg as ReceiveBeacon;
        interface SendMsg as SendDataPacket;
        interface ReceiveMsg as ReceiveDataPacket;
        interface SendMsg as SendTime;
        interface ReceiveMsg as ReceiveTime;
        interface SendMsg as SendIamNew;
    }
}

```

```

interface ReceiveMsg as ReceiveIamNew;
interface SendMsg as SendParent;
interface ReceiveMsg as ReceiveParent;
interface SendMsg as SendSecParent;
interface ReceiveMsg as ReceiveSecParent;
interface SendMsg as SendAcceptParent;
interface ReceiveMsg as ReceiveAcceptParent;
interface SendMsg as SendSecondaryParent;
interface ReceiveMsg as ReceiveSecondaryParent;
interface SendMsg as SendAcceptSecParent;
interface ReceiveMsg as ReceiveAcceptSecParent;
interface SendMsg as SendAck;
interface ReceiveMsg as ReceiveAck;
interface SendMsg as SendReplyAck;
interface ReceiveMsg as ReceiveReplyAck;
}
}
implementation{
    TOS_Msg Data_Packet;
    TOS_Msg Time_Packet;
    TOS_Msg Start_Packet;
    TOS_Msg New_Packet;
    TOS_Msg Parent_Packet;
    TOS_Msg Sec_Packet;
    TOS_Msg Accept_Packet;
    TOS_Msg Secondary_Packet;
    TOS_Msg Ack_Packet;
    TOS_Msg ReplyAck_Packet;

    uint16_t message_id;
    uint16_t destination, new_destination;
    uint16_t parent = 0xFFFF;
    uint16_t data_sent;
    uint16_t node_depth;
    uint16_t pmsg_destination;
    uint16_t accepted_destination, ack_dest;
    uint16_t sp_destination;
    uint16_t accepted_sec_dest, sec_parent = 0;
    uint16_t parent_depth, sec_parent_depth;
    uint16_t time_sent = 0;
    bool child, sec_child, fsd;

static void initialize() {
    sec_child = TRUE;
    child = TRUE;

```

```

    fsd = TRUE;
}

static void newdest() /* Function declaring destination addresses for each node */
{
    if (TOS_LOCAL_ADDRESS > 3)
    {
        new_destination = (TOS_LOCAL_ADDRESS - 3);
    }
    else
    {
        new_destination = TOS_BCAST_ADDR;
    }
}

/* Initializing nodes */
command result_t StdControl.init(){
    initialize();
    newdest();
    call Leds.init();
    call Leds.redOn();
    return SUCCESS;
}

command result_t StdControl.start()
/* A timer is fired at the lapse of 1s triggering the transmission of a beacon from the base
station*/
{
    if (TOS_LOCAL_ADDRESS == 0)
    {
        call RouteTimer.start(TIMER_ONE_SHOT,1000);
        call Leds.redOff();
    }
    return SUCCESS;
}

command result_t StdControl.stop()
{
    call RouteTimer.stop();
    call DelayTimer.stop();
    call AckTimer.stop();
    return SUCCESS;
}

/* Firing of Beacon at base station */

```

```

event result_t RouteTimer.fired(){
    if (TOS_LOCAL_ADDRESS == 0)
    {
        Beacon *a = (Beacon *) Start_Packet.data;
        a -> sourceaddress = TOS_LOCAL_ADDRESS;
        a -> dest = 1;
        a -> tree_depth = 0;
        call SendBeacon.send(TOS_BCAST_ADDR, sizeof(Beacon),
&Start_Packet);

    }
    return SUCCESS;
}

event result_t SendBeacon.sendDone(TOS_MsgPtr msg, bool success) /* Event
signalling successful completion of beacon transmission */
{
    return SUCCESS;
}

event TOS_MsgPtr ReceiveBeacon.receive(TOS_MsgPtr recv_packet)
{
    Beacon *b = (Beacon *)recv_packet -> data;

    if (b -> dest == TOS_LOCAL_ADDRESS)
    {
        call ADC.getData(); /* Request corresponding ADC to sample local sensors */
        parent = 0;
        node_depth = (b -> tree_depth) + 1;
        child = FALSE;
        fsd = FALSE;
        call Leds.redOff();
    }
    else
    {
        parent = 0;
        node_depth = (b -> tree_depth) + 1;
        child = FALSE;
        fsd = FALSE;
    }
    return recv_packet;
}

event result_t DelayTimer.fired()
{

```

```

    return call ADC.getData();
}

async event result_t ADC.dataReady(uint16_t data)
{
    if (!fsd)
    {
        DataMsg *d = (DataMsg *)Data_Packet.data;
        data_sent = (7-((data>>7) &0x7)); /* Disply MSB values of sensor readings */
        if (TOS_LOCAL_ADDRESS == 3)
        {
            destination = 1;
        }
        else if (TOS_LOCAL_ADDRESS == 6)
        {
            destination = 4;
        }
        else
        {
            destination = TOS_LOCAL_ADDRESS + 1;
        }
        d -> source = TOS_LOCAL_ADDRESS;
        d -> destinat = destination;
        d -> data = data_sent;
        call Leds.yellowOn();
        if (TOS_LOCAL_ADDRESS > 3)
        {
            if ((parent != 0xFFFF)&& (sec_parent != 0))
            {
                call SendDataPacket.send(TOS_BCAST_ADDR, sizeof(DataMsg),
&Data_Packet);
            }
        }
        else if ((TOS_LOCAL_ADDRESS <= 3) && (parent != 0xFFFF))
        {
            call SendDataPacket.send(TOS_BCAST_ADDR, sizeof(DataMsg),
&Data_Packet);
        }
        return SUCCESS;
    }
}

event result_t SendDataPacket.sendDone(TOS_MsgPtr msg, bool success)
{
    if (TOS_LOCAL_ADDRESS != 0)

```

```

    {
    TimeMsg *t = (TimeMsg *)Time_Packet.data;
    t -> source = TOS_LOCAL_ADDRESS;
    t -> parent = parent;
    t -> sec_parent = sec_parent;
    t -> tree_depth = node_depth;
    t -> id = message_id;
    t -> sent_destination = destination;
    t -> data = data_sent;
    t -> timex = msg -> time;
    message_id++;
    time_sent++;
    call SendTime.send(parent, sizeof(TimeMsg), &Time_Packet);
    call Leds.greenOn();
    }
    return SUCCESS;
}

event result_t SendTime.sendDone(TOS_MsgPtr msg, bool success)
{
    AckMsg *ack = (AckMsg *)Ack_Packet.data;
    ack -> source = TOS_LOCAL_ADDRESS;
    if ((time_sent >= 0x0008) && (parent != 0))
    {
        call SendAck.send(parent, sizeof(AckMsg), &Ack_Packet);
        call Leds.yellowOff();
        fsd = TRUE;
        call AckTimer.start(TIMER_REPEAT, 10000);
    }
    return SUCCESS;
}

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

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

event TOS_MsgPtr ReceiveAck.receive(TOS_MsgPtr recv_packet)
{
    AckMsg *ackr = (AckMsg *)recv_packet -> data;

```

```

ReplyAckMsg *rack = (ReplyAckMsg *)ReplyAck_Packet.data;
rack -> source = TOS_LOCAL_ADDRESS;
ack_dest = ackr -> source;
call SendReplyAck.send(ack_dest, sizeof(ReplyAckMsg), &ReplyAck_Packet);
return rcv_packet;
}

event TOS_MsgPtr ReceiveReplyAck.receive(TOS_MsgPtr rcv_packet)
{
    ReplyAckMsg *rackr = (ReplyAckMsg *)rcv_packet -> data;
    fsd = FALSE;
    return rcv_packet;
}

event result_t AckTimer.fired()
{
    SecParentMsg *sp = (SecParentMsg *)Sec_Packet.data;
    sp -> source = TOS_LOCAL_ADDRESS;
    if (fsd)
    {
        parent = sec_parent;
        node_depth = (parent_depth + 1);
        call SendSecParent.send(TOS_BCAST_ADDR, sizeof(SecParentMsg),
&Sec_Packet);
    }
    return SUCCESS;
}

event TOS_MsgPtr ReceiveDataPacket.receive(TOS_MsgPtr rcv_packet)
{
    DataMsg *e = (DataMsg *)rcv_packet -> data;
    DataMsg *d = (DataMsg *)Data_Packet.data;
    NewMsg *n = (NewMsg *)New_Packet.data;
    if ((e -> destinat == TOS_LOCAL_ADDRESS) && (TOS_LOCAL_ADDRESS !=
0))
    {
        call DelayTimer.start(TIMER_REPEAT,6000);
        call Leds.redOff();
    }
    else if ((TOS_LOCAL_ADDRESS != 0) && (parent == 0xFFFF))
    {
        n -> source = TOS_LOCAL_ADDRESS;
        call SendIamNew.send(new_destination, sizeof(NewMsg),
&New_Packet);
        call Leds.redOff();
    }
}

```



```

    }
    return recv_packet;
}

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

event TOS_MsgPtr ReceiveTime.receive(TOS_MsgPtr recv_packet)
{
    TimeMsg *t = (TimeMsg *)recv_packet -> data;
    TimeMsg *u = (TimeMsg *)Time_Packet.data;
    u -> source = t -> source;
    u -> parent = t -> parent;
    u -> sec_parent = t -> sec_parent;
    u -> tree_depth = t -> tree_depth;
    u -> id = t -> id;
    u -> sent_destination = t -> sent_destination;
    u -> data = t -> data;
    u -> timex = t -> timex;
    if (TOS_LOCAL_ADDRESS == 0)
    {
        call Leds.redToggle();
        call SendTime.send(TOS_UART_ADDR, sizeof(TimeMsg),
&Time_Packet);
    }
    else if ((TOS_LOCAL_ADDRESS != 0) && (parent != 0xFFFF))
    {
        call SendTime.send(parent, sizeof(TimeMsg), &Time_Packet);
    }
    return recv_packet;
}

event TOS_MsgPtr ReceiveIamNew.receive(TOS_MsgPtr recv_packet)
{
    NewMsg *n = (NewMsg *)recv_packet -> data;
    ParentMsg *pmsg = (ParentMsg *)Parent_Packet.data;
    pmsg -> source = TOS_LOCAL_ADDRESS;
    pmsg -> tree_depth = node_depth;
    pmsg_destination = n -> source;
    fsd = TRUE;
    call SendParent.send(pmsg_destination, sizeof(ParentMsg), &Parent_Packet);
    return recv_packet;
}

```

```

}

event result_t SendParent.sendDone(TOS_MsgPtr msg, bool success)
{
    fsd = FALSE;
    return SUCCESS;
}

event TOS_MsgPtr ReceiveParent.receive(TOS_MsgPtr recv_packet)
{
    ParentMsg *pmsgr = (ParentMsg *)recv_packet -> data;
    SecParentMsg *sp = (SecParentMsg *)Sec_Packet.data;
    AcceptMsg *am = (AcceptMsg *)Accept_Packet.data;
    atomic
    {
        parent = pmsgr -> source;
    }
    atomic
    {
        node_depth = (pmsgr -> tree_depth) + 1;
        parent_depth = pmsgr -> tree_depth;
    }
    sp -> source = TOS_LOCAL_ADDRESS;
    am -> source = TOS_LOCAL_ADDRESS;
    accepted_destination = pmsgr -> source;
    call Leds.redOn();
    call SendSecParent.send(TOS_BCAST_ADDR, sizeof(SecParentMsg),
&Sec_Packet);
    call SendAcceptParent.send(parent, sizeof(AcceptMsg), &Accept_Packet);
    fsd = TRUE;
    return recv_packet;
}

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

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

event TOS_MsgPtr ReceiveSecParent.receive(TOS_MsgPtr recv_packet)
{

```

```

if (TOS_LOCAL_ADDRESS != 0)
{
    SecParentMsg *spr = (SecParentMsg *)recv_packet ->data;
    SecparentMsg *Spm = (SecparentMsg *)Secondary_Packet.data;
    Spm -> source = TOS_LOCAL_ADDRESS;
    Spm -> tree_depth = node_depth;
    sp_destination = spr -> source;
    call SendSecondaryParent.send(sp_destination, sizeof(SecparentMsg),
&Secondary_Packet);
}
return recv_packet;
}

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

event TOS_MsgPtr ReceiveAcceptParent.receive(TOS_MsgPtr recv_packet)
{
    if (TOS_LOCAL_ADDRESS != 0)
    {
        AcceptMsg *amr = (AcceptMsg *)recv_packet -> data;
        call Leds.greenOff();
        child = TRUE;
        fsd = FALSE;
        call ADC.getData();
    }
    return recv_packet;
}

event TOS_MsgPtr ReceiveSecondaryParent.receive(TOS_MsgPtr recv_packet)
{
    SecParentMsg *Spmr = (SecParentMsg *)recv_packet -> data;
    AcceptSecMsg *amsp = (AcceptSecMsg *) Accept_Packet.data;
    atomic
    {
        sec_parent = Spmr -> source;
        sec_parent_depth = Spmr -> tree_depth;
    }
    amsp -> source = TOS_LOCAL_ADDRESS;
    accepted_sec_dest = Spmr -> source;
    call Leds.greenOn();
    if ((sec_parent != 0) && (sec_parent != parent))
    {

```

```

        call SendAcceptSecParent.send(accepted_sec_dest, sizeof(AcceptSecMsg),
&Accept_Packet);
    }
    return recv_packet;
}

event TOS_MsgPtr ReceiveAcceptSecParent.receive(TOS_MsgPtr recv_packet)
{
    AcceptSecMsg *amr = (AcceptSecMsg *)recv_packet;
    sec_child = TRUE;
    return recv_packet;
}

event result_t SendAcceptSecParent.sendDone(TOS_MsgPtr msg, bool success)
{
    fsd = FALSE;
    call ADC.getData();
    return SUCCESS;
}
}

```

APPENDIX B

JAVA AND PERL CODES

This appendix consists of the Java and Perl code that is used to receive the messages from various nodes at the base station and write into a text file which can be used later to study the sequence of events occurring in the network. The perl code is used to create a user interface that can be used to query the text file about the events that have been recorded.

TimeMsg.java:

The following java code is responsible for receiving the messages from the nodes through the USB port of the base station and displaying them on the screen.

```
package net.tinyos.Ad_hoc;

public class TimeMsg extends net.tinyos.message.Message {

    /** The default size of this message type in bytes. */
    public static final int DEFAULT_MESSAGE_SIZE = 16;

    /** The Active Message type associated with this message. */
    public static final int AM_TYPE = 42;

    /** Create a new TimeMsg of size 16. */
    public TimeMsg() {
        super(DEFAULT_MESSAGE_SIZE);
        amTypeSet(AM_TYPE);
    }
    /** Create a new TimeMsg of the given data_length. */
```

```

public TimeMsg(int data_length) {
    super(data_length);amTypeSet(AM_TYPE);
} /**
 * Create a new TimeMsg with the given data_length
 * and base offset.

 */
public TimeMsg(int data_length, int base_offset) {
    super(data_length, base_offset);
    amTypeSet(AM_TYPE);
}

/**
 * Create a new TimeMsg using the given byte array
 * as backing store.
 */
public TimeMsg(byte[] data) {
    super(data);
    amTypeSet(AM_TYPE);
}

/**
 * Create a new TimeMsg using the given byte array
 * as backing store, with the given base offset.
 */
public TimeMsg(byte[] data, int base_offset) {
    super(data, base_offset);
    amTypeSet(AM_TYPE);
}

/**
 * Create a new TimeMsg using the given byte array
 * as backing store, with the given base offset and data length.
 */
public TimeMsg(byte[] data, int base_offset, int data_length) {
    super(data, base_offset, data_length);
    amTypeSet(AM_TYPE);
}

/**
 * Create a new TimeMsg embedded in the given message
 * at the given base offset.
 */
public TimeMsg(net.tinyos.message.Message msg, int base_offset) {
    super(msg, base_offset, DEFAULT_MESSAGE_SIZE);
}

```

```

        amTypeSet(AM_TYPE);
    }

/**
 * Create a new TimeMsg embedded in the given message
 * at the given base offset and length.
 */
public TimeMsg(net.tinyos.message.Message msg, int base_offset, int data_length) {
    super(msg, base_offset, data_length);
    amTypeSet(AM_TYPE);
}

/**
 * Return a String representation of this message. Includes the
 * message type name and the non-indexed field values.
 */
public String toString() {
    String s = "Message <TimeMsg> \n";
    try {
        s += " [source=0x"+Long.toHexString(get_source())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [parent=0x"+Long.toHexString(get_parent())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [sec_parent=0x"+Long.toHexString(get_sec_parent())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [tree_depth=0x"+Long.toHexString(get_tree_depth())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [id=0x"+Long.toHexString(get_id())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [sent_destination=0x"+Long.toHexString(get_sent_destination())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [data=0x"+Long.toHexString(get_data())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    try {
        s += " [timex=0x"+Long.toHexString(get_timex())+"]\n";
    } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip field */ }
    return s;
}

```

```

// Message-type-specific access methods appear below.

////////////////////////////////////
// Accessor methods for field: source
// Field type: int, unsigned
// Offset (bits): 0
// Size (bits): 16
////////////////////////////////////

/**
 * Return whether the field 'source' is signed (false).
 */
public static boolean isSigned_source() {
    return false;
}

/**
 * Return whether the field 'source' is an array (false).
 */
public static boolean isArray_source() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'source'
 */
public static int offset_source() {
    return (0 / 8);
}

/**
 * Return the offset (in bits) of the field 'source'
 */
public static int offsetBits_source() {
    return 0;
}

/**
 * Return the value (as a int) of the field 'source'
 */
public int get_source() {
    return (int)getUIntElement(offsetBits_source(), 16);
}

/**

```



```

    * Set the value of the field 'source'
    */
public void set_source(int value) {
    setUIntElement(offsetBits_source(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'source'
 */
public static int size_source() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'source'
 */
public static int sizeBits_source() {
    return 16;
}

////////////////////////////////////
// Accessor methods for field: parent
// Field type: int, unsigned
// Offset (bits): 16
// Size (bits): 16
////////////////////////////////////

/**
 * Return whether the field 'parent' is signed (false).
 */
public static boolean isSigned_parent() {
    return false;
}

/**
 * Return whether the field 'parent' is an array (false).
 */
public static boolean isArray_parent() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'parent'
 */
public static int offset_parent() {

```

```

    return (16 / 8);
}

/**
 * Return the offset (in bits) of the field 'parent'
 */
public static int offsetBits_parent() {
    return 16;
}

/**
 * Return the value (as a int) of the field 'parent'
 */
public int get_parent() {
    return (int)getUIntElement(offsetBits_parent(), 16);
}

/**
 * Set the value of the field 'parent'
 */
public void set_parent(int value) {
    setUIntElement(offsetBits_parent(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'parent'
 */
public static int size_parent() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'parent'
 */
public static int sizeBits_parent() {
    return 16;
}

////////////////////////////////////
// Accessor methods for field: sec_parent
// Field type: int, unsigned
// Offset (bits): 32
// Size (bits): 16
////////////////////////////////////

```

```

/**
 * Return whether the field 'sec_parent' is signed (false).
 */
public static boolean isSigned_sec_parent() {
    return false;
}

/**
 * Return whether the field 'sec_parent' is an array (false).
 */
public static boolean isArray_sec_parent() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'sec_parent'
 */
public static int offset_sec_parent() {
    return (32 / 8);
}

/**
 * Return the offset (in bits) of the field 'sec_parent'
 */
public static int offsetBits_sec_parent() {
    return 32;
}

/**
 * Return the value (as a int) of the field 'sec_parent'
 */
public int get_sec_parent() {
    return (int)getUIntElement(offsetBits_sec_parent(), 16);
}

/**
 * Set the value of the field 'sec_parent'
 */
public void set_sec_parent(int value) {
    setUIntElement(offsetBits_sec_parent(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'sec_parent'
 */

```

```

public static int size_sec_parent() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'sec_parent'
 */
public static int sizeBits_sec_parent() {
    return 16;
}

////////////////////////////////////
// Accessor methods for field: tree_depth
// Field type: int, unsigned
// Offset (bits): 48
// Size (bits): 16
////////////////////////////////////

/**
 * Return whether the field 'tree_depth' is signed (false).
 */
public static boolean isSigned_tree_depth() {
    return false;
}

/**
 * Return whether the field 'tree_depth' is an array (false).
 */
public static boolean isArray_tree_depth() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'tree_depth'
 */
public static int offset_tree_depth() {
    return (48 / 8);
}

/**
 * Return the offset (in bits) of the field 'tree_depth'
 */
public static int offsetBits_tree_depth() {
    return 48;
}

```

```

/**
 * Return the value (as a int) of the field 'tree_depth'
 */
public int get_tree_depth() {
    return (int)getUIntElement(offsetBits_tree_depth(), 16);
}

/**
 * Set the value of the field 'tree_depth'
 */
public void set_tree_depth(int value) {
    setUIntElement(offsetBits_tree_depth(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'tree_depth'
 */
public static int size_tree_depth() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'tree_depth'
 */
public static int sizeBits_tree_depth() {
    return 16;
}

////////////////////////////////////
// Accessor methods for field: id
// Field type: int, unsigned
// Offset (bits): 64
// Size (bits): 16
////////////////////////////////////

/**
 * Return whether the field 'id' is signed (false).
 */
public static boolean isSigned_id() {
    return false;
}

/**
 * Return whether the field 'id' is an array (false).

```

```

*/
public static boolean isArray_id() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'id'
 */
public static int offset_id() {
    return (64 / 8);
}

/**
 * Return the offset (in bits) of the field 'id'
 */
public static int offsetBits_id() {
    return 64;
}

/**
 * Return the value (as a int) of the field 'id'
 */
public int get_id() {
    return (int)getUIntElement(offsetBits_id(), 16);
}

/**
 * Set the value of the field 'id'
 */
public void set_id(int value) {
    setUIntElement(offsetBits_id(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'id'
 */
public static int size_id() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'id'
 */
public static int sizeBits_id() {
    return 16;
}

```

```

}

////////////////////////////////////
// Accessor methods for field: sent_destination
// Field type: int, unsigned
// Offset (bits): 80
// Size (bits): 16
////////////////////////////////////

/**
 * Return whether the field 'sent_destination' is signed (false).
 */
public static boolean isSigned_sent_destination() {
    return false;
}

/**
 * Return whether the field 'sent_destination' is an array (false).
 */
public static boolean isArray_sent_destination() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'sent_destination'
 */
public static int offset_sent_destination() {
    return (80 / 8);
}

/**
 * Return the offset (in bits) of the field 'sent_destination'
 */
public static int offsetBits_sent_destination() {
    return 80;
}

/**
 * Return the value (as a int) of the field 'sent_destination'
 */
public int get_sent_destination() {
    return (int)getIntElement(offsetBits_sent_destination(), 16);
}

/**

```

```

    * Set the value of the field 'sent_destination'
    */
public void set_sent_destination(int value) {
    setUIntElement(offsetBits_sent_destination(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'sent_destination'
 */
public static int size_sent_destination() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'sent_destination'
 */
public static int sizeBits_sent_destination() {
    return 16;
}

////////////////////////////////////
// Accessor methods for field: data
// Field type: int, unsigned
// Offset (bits): 96
// Size (bits): 16
////////////////////////////////////

/**
 * Return whether the field 'data' is signed (false).
 */
public static boolean isSigned_data() {
    return false;
}

/**
 * Return whether the field 'data' is an array (false).
 */
public static boolean isArray_data() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'data'
 */
public static int offset_data() {

```



```

    return (96 / 8);
}

/**
 * Return the offset (in bits) of the field 'data'
 */
public static int offsetBits_data() {
    return 96;
}

/**
 * Return the value (as a int) of the field 'data'
 */
public int get_data() {
    return (int)getUIntElement(offsetBits_data(), 16);
}

/**
 * Set the value of the field 'data'
 */
public void set_data(int value) {
    setUIntElement(offsetBits_data(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'data'
 */
public static int size_data() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'data'
 */
public static int sizeBits_data() {
    return 16;
}

////////////////////////////////////
// Accessor methods for field: timex
// Field type: int, unsigned
// Offset (bits): 112
// Size (bits): 16
////////////////////////////////////

```

```

/**
 * Return whether the field 'timex' is signed (false).
 */
public static boolean isSigned_timex() {
    return false;
}

/**
 * Return whether the field 'timex' is an array (false).
 */
public static boolean isArray_timex() {
    return false;
}

/**
 * Return the offset (in bytes) of the field 'timex'
 */
public static int offset_timex() {
    return (112 / 8);
}

/**
 * Return the offset (in bits) of the field 'timex'
 */
public static int offsetBits_timex() {
    return 112;
}

/**
 * Return the value (as a int) of the field 'timex'
 */
public int get_timex() {
    return (int)getUIntElement(offsetBits_timex(), 16);
}

/**
 * Set the value of the field 'timex'
 */
public void set_timex(int value) {
    setUIntElement(offsetBits_timex(), 16, value);
}

/**
 * Return the size, in bytes, of the field 'timex'
 */

```

```
public static int size_timex() {
    return (16 / 8);
}

/**
 * Return the size, in bits, of the field 'timex'
 */
public static int sizeBits_timex() {
    return 16;
}
}
```

The following code is used to write the messages received at the base station into a text file, log.txt.

```
package net.tinyos.Ad_hoc;

import java.io.*;
import net.tinyos.message.*;
import net.tinyos.packet.*;

public class TimeMsgIF implements MessageListener {
//int a,a_1,a_2,a_3,a_4,a_5,a_6,a_7 = 0;
//int s,s_1,s_2,s_3,s_4,s_5,s_6,s_7 = 0;
//int packets_lost;
int message_id = 0;
private static final String LOG_FILENAME = "log.txt";

PrintWriter log_os;

public TimeMsgIF() {

    try{
        //create a file for logging data to.
        FileOutputStream f = new FileOutputStream("log.txt");
        log_os = new PrintWriter(f);
    } catch (Exception e) {
        e.printStackTrace();
    }

    // This connects to the SerialForwarder running on the local node.
    MoteIF moteif = new MoteIF((net.tinyos.util.Messenger)null);

    // Listen for messages of type MultihopMsg (generated by the 'mig'
    // step above).
    moteif.registerListener(new TimeMsg(), this);
}
```

Parse.pl

The following perl code is responsible for creating the interface that is used to query the text file containing the messages.

```
#####  
#####  
# PARSE LOG FILE FOR FOLLOWING FIELDS  
#STRUCTURE ON FILE          EXTRACTED  
#TimeMsg:Message <TimeMsg>  
# [source=0x3]              - SOURCE  
# [parent=0x0]              - PARENT  
# [sec_parent=0x0]  
# [tree_depth=0x1]  
# [id=0xa]                  - ID  
# [sent_destination=0x1]  
# [data=0x7]  
# [timex=0xc5]              - TIMEX  
  
#####  
#####  
# INITIALIZATIONS  
#####  
#####  
# PATH VARIABLE FOR SCRIPT  
my $file_name = "C:\\Program Files\\UCB\\cygwin\\opt\\tinyos-  
1.x\\tools\\java\\net\\tinyos\\Ad_hoc\\log.txt";  
  
# VARIABLES  
my $Msg_counter,$src,$dest;  
my %src_list;  
  
my $TimeMsg_counter,$source, $parent, $id, $timex;  
my $destination;  
my %id_list, %timex_list, %source_parents,%parent_change_track;  
  
my $prev_parent = -1;  
my $curr_parent = -1;  
my $parent_change = 0;  
my $error = 0;
```

```

*****
*****
# PRE_READ TO POPULATE OPTIONS
*****
*****
open(PRE_READ,$file_name) || die("Could not open file!");
while(<PRE_READ>)
{
  chomp;
  my @field = split /\t/;
  my $field1 = $field[0];

  # EXTRACT MESSAGE ID
  if($field1 =~ /TimeMsg\w*/)
  { my @msg_id_arr = split(" ", $field1);
    $Msg_counter = $msg_id_arr[0];
  }

  # EXTRACT SOURCE
  elsif($field1 =~ /source\=\d+\/)
  { $src = substr($',1, 1); }

  # EXTRACT DESTINATION
  elsif($field1 =~ /sent_destination\=\d+\/)
  { $dest = substr($',1, 1);
    if(exists ($src_list{$src}))
    { if($src_list{$src} ne $dest) {print "PROBLEM MSG:$Msg_counter SRC:$src
$OLD_DEST:$src_list{$src} DEST:$dest\n "; }
      else
      { $src_list{$src}=$dest; }
    }
  }
}
close (PRE_READ);
#foreach my $key (sort keys %src_list) { print "$key--$src_list{$key}\n"; }

*****
*****
# GET USER INPUT
*****
*****
# INPUT SOURCE
system(cls);
print "\nSELECT THE SOURCE FROM THE LIST (";
foreach my $key (sort keys %src_list) { print "$key,"; }

```

```

print "):";
my $source_input = <STDIN>;
chomp($source_input);

# INPUT MESSAGE ID
print "Enter Starting Message ID < $Msg_counter:";
my $msg_id = <STDIN>;
chomp($msg_id);

# VALIDATE INPUTS
if($msg_id>$Msg_counter || not(exists($src_list{$source_input})))
{ $error = 1;
  print "\n\nINVALID ENTRY !!! \n";}

*****
*****
# ACTUAL PROGRAM
*****
*****
$destination = $src_list{$source_input};

open(DAT,$file_name) || die("Could not open file!");
while(<DAT>)
{
  chomp;
  my @field = split /\t/;
  $field1 = $field[0];
  # EXTRACT MESSAGE ID
  if($field1 =~ /TimeMsg\w*/)
  {
    my @msg_id_arr = split(" ", $field1);
    $TimeMsg_counter = $msg_id_arr[0];
  }

  # EXTRACT SOURCE
  elsif($field1 =~ /source\=\d+\/)
  { $source = substr($',1, 1); }

  # EXTRACT PARENT
  elsif($field1 =~ /sec_parent\=\d+\/)
  { }
  elsif($field1 =~ /parent\=\d+\/)
  {
    $curr_parent = substr($',1, 1);

```

```

if ($source eq $source_input && $TimeMsg_counter >= $msg_id)
{
    if ($prev_parent eq "-1") {$prev_parent = $curr_parent}
    else
    {
        if($curr_parent eq $prev_parent) {}
        else
        {
            $parent_change++;
            #print "Message ID: $TimeMsg_counter -> Parent
Changed from $prev_parent to $curr_parent \n";

            $parent_change_track{$TimeMsg_counter}{"CURRENT_PARENT"} =
$curr_parent;

            $parent_change_track{$TimeMsg_counter}{"PREVIOUS_PARENT"} =
$prev_parent;

                                $prev_parent = $curr_parent;
                                }
                                }
        }
    }
}

# EXTRACT ID
elseif($field1 =~ /id\w*/)
{
my $svar = $';
chop($svar);
my @fsields = split(/x/,$svar);
$Sid = $fsields[1];

if ($source eq $destination && $TimeMsg_counter >= $msg_id )
{ $Sid_list{$TimeMsg_counter} = $Sid; }
}

# EXTRACT TIMEX INFORMATION
elseif($field1 =~ /timex\w*/)
{
my $svar = $';
chop($svar);
my @fsields = split(/x/,$svar);
$timex = $fsields[1];

```



```

    if ($source eq $destination && $TimeMsg_counter >= $msg_id)
    { $timex_list{$TimeMsg_counter} = $timex; }
}
else {}

}
close (DAT);

#####
#####
# PRINTS OUTPUT DATA TO A FILE
#####
#####
open(OUT,"+>". "Output.txt") || die("Could not open file!");

# PRINT RUN INFORMATION
print OUT "RESULTS LOG \n";
{
    use Time::localtime;
    my $tm = localtime;
    my $year = $tm->year+1900;my $month = ($tm->mon)+1;my $day = $tm->mday;
    print OUT "Date: $month-$day-$year\n";
}
print OUT "Input File: $file_name\n";

# PRINT FINAL EXTRACTED DATA
print OUT
"\n#####
#####";
print OUT
"\n#####
#####";
if ($error ne 1)
{
    print OUT "\n\n SOURCE    : $source_input";
    print OUT "\n\n DESTINATION : $destination";
    print OUT "\n\n CHANGE IN PARENTS\n";
    foreach my $key (sort keys %parent_change_track)
    { my $curr = $parent_change_track{$key}{"CURRENT_PARENT"};
      my $prev = $parent_change_track{$key}{"PREVIOUS_PARENT"};
      print OUT " MSG ID:$key -> Parent Changed from $prev to $curr \n";
    }
    print OUT "\n Total Number of Parent Changes: $parent_change";
    print OUT "\n\n IDs AND TIMEX FOR DESTINATION\n";
}

```

```

        foreach my $key (sort keys %id_list) { print OUT " MSG ID:$key
ID:$id_list{$key} TIMEX:$timex_list{$key} \n"; }
        print "\n\n... Completed"
    }

# ERROR HANDLING
else
{
    print OUT "\n\nWrong inputs were entered\n\n";
}
print OUT
"\n*****";
print OUT
"\n*****";

close (OUT);

```