SIMULATION OF A MOBILE AGENT MIDDLEWARE FOR WIRELESS SENSOR NETWORKS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

SÜLEYMAN ÖZARSLAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

JANUARY 2008

Approval of the Graduate School of Informatics

_____
Prof. Dr. Nazife BAYKAL
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____
Assoc. Prof. Dr. Yasemin YARDIMCI
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Yasemin YARDIMCI Supervisor	Assoc. Prof. Dr. Y. Murat ERTEN Co- Supervisor

Examining Committee Members

Prof. Dr. Semih BİLGEN	(METU, EE)	_____
Assoc. Prof. Dr. Yasemin YARDIMCI	(METU, II)	_____
Dr. Erhan EREN	(METU, II)	_____
Assoc. Prof. Dr. Y. Murat ERTEN	(TOBB ETU, CENG)	_____
Dr. Alptekin TEMİZEL	(METU, II)	_____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Süleyman ÖZARSLAN

Signature:

ABSTRACT

SIMULATION OF A MOBILE AGENT MIDDLEWARE
FOR WIRELESS SENSOR NETWORKS

Özarslan, Süleyman

M.S., Department of Information Systems

Supervisor: Assoc. Prof. Dr. Yasemin Yardımcı

Co-Supervisor: Assoc. Prof. Dr. Y. Murat Erten

January 2008, 89 pages

Wireless Sensor Networks (WSNs) have become a significant research area in recent years as they play an increasingly important role in bridging the gap between the physical and the virtual world. However, programming wireless sensor networks is extremely challenging. Middleware for WSNs supports the development of sensing-based applications and facilitates programming wireless sensor networks. Middleware is the software that sits in the middle of applications and operating system. There is a large amount of research on middleware development for WSNs.

In this thesis, a first attempt is made to simulate mobile agents running on Agilla middleware using a software application. A Java application (Agilla Simulator) is developed and different agents corresponding to various functions are simulated. The performance of the network, namely the time it takes to execute the agents on the simulator is measured. The results of migration delay and reliability in the simulations of different agents are compared with those of the real world experiments. The comparison results presented in the study show that simulations produce results comparable to real life experiments.

Keywords: Wireless sensor networks, middleware, simulation, mobile agents

ÖZ

TELSİZ ALGILAYICI AĞLARI İÇİN

BIR GEZGIN ETMEN ORTAKATMAN YAZILIMININ BENZETİMİ

Özarslan, Süleyman

Yüksek Lisans, Bilişim Sistemleri

Tez Yöneticisi: Doç. Dr. Yasemin Yardımcı

Yardımcı Tez Yöneticisi: Doç. Dr. Y. Murat Erten

Ocak 2008, 89 sayfa

Fiziksel dünya ve sanal dünya arasında boşluğu doldurmada gittikçe daha da önemli bir rol oynayan telsiz algılayıcı ağları (TAA), son yıllarda önemli bir araştırma alanı haline gelmiştir. Bununla birlikte, telsiz algılayıcı ağlarının programlanması son derece zordur. TAA için geliştirilen ortakatman yazılımları, algılama tabanlı uygulamaların geliştirilmesini destekler ve telsiz algılayıcı ağlarının programlanmasını kolaylaştırır. Ortakatman, uygulamalar ve işletim sistemlerinin ortasında yer alan bir yazılımdır. TAA için ortakatman yazılımı geliştirilmesi üzerine çok sayıda araştırma yapılmaktadır.

Bu tezde, Agilla ortakatman yazılımı üzerinde çalışan gezgin etmenlerin bir yazılım uygulaması kullanılarak benzetimi ilk kez denenmiştir. Bir Java uygulaması (Agilla Benzetimcisi) geliştirilmiş ve çeşitli fonksiyonlara karşılık gelen farklı etmenlerin benzetimi yapılmıştır. Ağın başarımı, diğer bir deyişle etmenlerin benzetimcide çalıştırılması için gereken zaman ölçülmüştür. Değişik etmenlerin benzetimlerdeki taşınma zamanı ve güvenirlik sonuçları gerçek hayat deneylerinin sonuçlarıyla karşılaştırılmıştır. Bu çalışmada sunulan karşılaştırma sonuçları göstermiştir ki benzetimler gerçek hayat deneyleriyle benzer sonuçlar üretmektedir.

Anahtar Kelimeler: Telsiz algılayıcı ağları, ortakatman yazılımı, benzetim, gezgin etmenler

To My Family

ACKNOWLEDGMENTS

First of all, I would like to thank my adviser Assoc. Prof. Dr. Y. Murat Erten for his inexhaustible support, guidance and patience throughout the project. I am very grateful for his inspiring ideas.

I owe special thanks to my supervisor Assoc. Prof. Dr. Yasemin Yardımcı for her valuable suggestions.

I am grateful to Ayışığı for diligent proofreading my thesis; Pinar for sharing her experiences and Ufuk for practicing the talks with me. I also wish to thank other colleagues at the institute for an excellent working atmosphere.

Finally, I thank so much to my family. Especially, I am very grateful to my beloved wife Eylem, for her endless love, patience and support during the thesis period. The birth of our son was the best experience that we lived in this period. He adds a new dimension to our lives. I would like to thank my parents, my brother Can and my sister Gamze for their endless love.

TABLE OF CONTENTS

ABSTRACT…… viii

ÖZ……. viii

DEDICATION viii

ACKNOWLEDGMENTS ix

TABLE OF CONTENTS x

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS xvi

CHAPTER

1. INTRODUCTION 1

1.1. BACKGROUND OF THE PROBLEM 1

1.2. PURPOSE OF THE STUDY 3

1.3. OUTLINE OF THE THESIS 4

2. WIRELESS SENSOR NETWORKS AND MIDDLEWARE 6

2.1. WIRELESS SENSOR NETWORKS 6

2.1.1. WSN Challenges and Current Research Areas 7

2.2. HARDWARE PLATFORMS 8

2.2.1. Mica Family Motes 8

2.2.2. MoteIV Motes 10
2.3. MIDDLEWARE FOR WIRELESS SENSOR NETWORKS 11
2.3.1. Middleware Challenges 12
2.4. MIDDLEWARE APPROACHES 14
2.4.1. Database Approach 15
2.4.2. Virtual Machine Approach 18
2.4.3. Adaptive Approach 20
2.4.4. Agent Based Approach 21
2.5. ADVANTAGES OF MOBILE AGENT MIDDLEWARE 23
3. SOFTWARE ARCHITECTURE 25
3.1. TINYOS 25
3.2. TOSSIM 26
3.3. TINYVIZ 29
3.4. SERIAL FORWARDER 31
3.5. AGILLA 32
3.5.1. Agilla Model 32
3.5.2. Implementation Platform of Agilla 36
3.5.3. Agilla Architecture 36
4. MODIFICATIONS ON AGILLA 42
4.1. AGILLA SIMULATOR 42
4.1.1. Opening or Creating an Agent 43
4.1.2. Converting the Agent's Code to NesC Code 43
4.1.3. Embedding the Converted Code 44
4.1.4. Compiling Agilla 44
4.1.5. Launching Serial Forwarder (SF) and Connecting TOSSIM to SF. 45

4.1.6. Running Simulation with TinyViz and Saving Log Files 45
4.1.7. Advantages Over the Simulation Method Described in the Agilla Website 46
4.2. IMPORTANT BUG FIXES 47
4.3. ADDED INSTRUCTIONS 47
5. SIMULATION STUDIES 49
5.1. BENEFITS OF SIMULATIONS 49
5.2. THE SIMULATION ENVIRONMENT 50
5.2.1. Network Topology 51
5.2.2. Routing 52
5.2.3. Radio Model 52
5.2.4. Simulation Parameters 52
5.3. SIMULATIONS 53
5.3.1. Benchmarking Performance of Important Instructions 53
5.3.2. Comparison of Simulations with Real Life WSN Experiments 64
6. CONCLUSIONS 71
6.1. SUMMARY OF THE WORK DONE 71
6.2. FURTHER RESEARCH 72
REFERENCES 74
APPENDICES 80
A. SAMPLE TINYVIZ AUTORUN FILES USED IN SIMULATIONS 80
B. IMPORTANT BUG FIXES 82
C. SETTING UP THE SIMULATION ENVIRONMENT 86

LIST OF TABLES

Table 2.1: Comparison of Middleware Approaches 14

Table 4.1: Added instructions to Agilla's ISA 48

Table 5.1: Line by Line Explanation of the Agent in Figure 5.9. 59

Table 5.2: Min, max and average values and variances for smove agent 66

Table 5.3: Min, max and average values and variances for rout agent 68

Table 5.4: Min, max and average values and variances for remote ops..... 70

LIST OF FIGURES

Figure 2.1: A Wireless Sensor Network 7

Figure 2.2: Diagram of Mica Mote 9

Figure 2.3: Mica and Mica2dot Motes 10

Figure 2.4: Tmote Sky mote 11

Figure 2.5: Common Architecture of Middleware 12

Figure 2.6: TinyDB GUI 17

Figure 3.1: TOSSIM Architecture 27

Figure 3.2: TinyViz GUI 29

Figure 3.3: Serial Forwarder GUI 31

Figure 3.4: Agilla's System Architecture 32

Figure 3.5: Tuple space operations 35

Figure 3.6: Agilla Architecture 37

Figure 3.7: The Mobile Agent Architecture 40

Figure 4.1: Agilla Simulator GUI 43

Figure 5.1: Grid topology of simulated system 51

Figure 5.2: The smove agent with heap operations 514

Figure 5.3: The wmove agent with heap operations 514

Figure 5.4: Latency of smove and wmove instructions with heap operations 55
Figure 5.5: The smove agent without heap operations. 56
Figure 5.6: The wmove agent without heap operations. 56
Figure 5.7: Latency of smove and wmove instructions without heap operations. 56
Figure 5.8: Latency of sclone and wclone instructions with heap operations 57
Figure 5.9: Spreading agent 578
Figure 5.10: The Agent's Spreading Time into WSN..... 60
Figure 5.11: Reliability equation of the agent..... 61
Figure 5.12: Reliability of the agent in Figure 5.8. 61
Figure 5.13: The Modified Agent's Spreading Time into WSN..... 62
Figure 5.14: Comparison Chart of Spreading Times 63
Figure 5.15: Reliability Comparison of Agents 63
Figure 5.16: 5x5 mote test bed which is used in real experiments 64
Figure 5.17: Code of smove agent 65
Figure 5.18: Comparison of simulations and real experiments results of latency of smove instruction. 65
Figure 5.19: Reliability formula of an instruction 66
Figure 5.20: Comparison of the results of simulations and real experiments of reliability of smove instruction. 67
Figure 5.21: Code of rout agent 68
Figure 5.22: Comparison of the results of simulations and real experiments of latency of rout instruction 68
Figure 5.23: Comparison of simulations and real experiments results of reliability of rout instruction..... 69
Figure 5.24: Comparison of results of the simulations and real experiments of latency of remote operations. 70

LIST OF A BBREVIATIONS

A/D	Analog to Digital
ADC	Analog-Digital Converter
AutoSeC	Automatic Service Composition
AWK	Aho, Weinberger, Kernighan
CPU	Central Processing Unit
CVS	Concurrent Versions System
DoS	Denial of Service
DSWare	Data Service Middleware
EEPROM	Electrically Erasable Programmable Read-Only Memory
Graphviz	Graph Visualization Software
GUI	Graphical User Interface
h	Hour
I^2C	Intelligent Interface Controller
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
ISA	Instruction Set Architecture
ISM	Industrial, Scientific and Medical
JDK	Java Development Kit
JVM	Java Virtual Machine
KB	Kilobyte
KHz	Kilohertz
KM	Kilometer

Kbit	Kilobit
LDR	Light Dependent Resistor
LED	Light Emitting Diode
LQI	Link Quality Indicator
m	Meter
MacOS	Macintosh Operating System
MARS	Mobile Agent Reactive Spaces
Max	Maximum
MHz	Megahertz
MiLAN	Middleware Linking Applications and Networks
Min	Minimum
mm	Millimeter
MPR	Mote Processor/Radio
ms	Millisecond
MTS	Mote Sensor
NesC	Network Embedded Systems C
NS-2	Network Simulator - 2
NTC	Negative Temperature Coefficient
OMNET++	Objective Modular Network Testbed in C++
PC	Personal Computer
PC	Program Counter
PDA	Personal Digital Assistant
PWM	Pulse Width Modulators
QoS	Quality of Service
RFM	Radio Frequency Module
RTS	Remote Tuple Space
sec	Second
SF	Serial Forwarder
SINA	Sensor Information Networking Architecture
SM	Smart Message
SQL	Structured Query Language

SQTL	Sensor Query and Tasking Language
SRAM	Static Random Access Memory
SSI	Single System Image
TACOMA	Tromsø and Cornell Moving Agents
TCP	Transport Control Protocol
TinyDB	Tiny Database
TinyOS	Tiny Microthreading Operating System
TinyViz	Tiny Visualization Software
TOSSIM	Tiny Microthreading Operating System Simulator
UC	University of California
USB	Universal Serial Bus
UART	Universal Asynchronous Receiver and Transmitter
V	Volt
VM	Virtual Machine
WSN	Wireless Sensor Network

CHAPTER 1

INTRODUCTION

This introductory chapter addresses the motivation of the study including background of the problem, purpose and significance of the study and finally, the outline followed throughout this thesis report.

1.1. BACKGROUND OF THE PROBLEM

The concept of sensor networks is fairly new and the first article on the wireless sensor network (WSN) concept was published in 1998 (Asada et al., 1998). A WSN is a networked collection of *sensor nodes* which are small-scale devices and have very limited resources such as computing power, memory, bandwidth and power supply (Akyildiz, Su, Sankarasubramaniam & Cayirci, 2002). Each sensor node has various sensors to detect some physical conditions such as light, sound, humidity, temperature, motion, pressure and vibration. In this thesis, the words sensor node and mote are used analogously.

As a result of this variety, WSNs support a wide range of applications. Arora et al. (2004) proposed a WSN for security and defense applications while Shen, Wang and Sun (2004) used WSNs for industrial automation. Some researchers also studied WSNs for habitat monitoring (Mainwaring, Culler, Polastre, Szewczyk &Anderson, 2002). Another application is a traffic control system demonstrated by Wenjie, Lifeng, Zhanglong and Shiliang (2005).

Malan, Fulford-Jones, Welsh and Moulton (2004) offer a healthcare application using WSNs. One final example is a fire tracking application established on a WSN is demonstrated by Fok, Roman and Lu (2005).

WSNs pose various challenges such as limited resources of motes, heterogeneity of nodes, network deployment, communication failures, security, routing, and programming. According to Boulis, Han and Srivastava (2003), programming wireless sensor networks is one of the most important challenges of WSNs. A middleware layer can help in the design and programming challenges of WSNs by bridging the gap between WSN applications and low level layers such as hardware and operating system.

Middleware can be classified considering their programming approaches: database approach, virtual machine approach, adaptive approach and agent based approach. There are important challenges which must be figured out by a successful middleware, such as ease of use, adaptability, managing resources, heterogeneity, security and quality of service (QoS). Each approach has strong points and weak points to cope with these challenges. A middleware inherits strong and weak characteristics of its approach, and it has also specific advantages and disadvantages.

Agent based approach is considered to be the most advantageous one usually. Mobile agents are used in this approach and they are dynamic, localized and intelligent programs that can move or clone across nodes to perform a specific task (Harrison, David & Kershenbaum, 1995). They are injected to the WSN, after which, they can distribute themselves through the network to perform their assigned tasks.

Agilla is the first mobile agent middleware for WSNs (Fok et al., 2005). It is a flexible, powerful and easy to use middleware application and it is considered to be an exemplary middleware because of its advantages. However, some improvements can be made in Agilla.

Until now, Agilla was used for different applications. Some of them are fire tracking (Fok et al., 2005), cargo tracking (Hackmann et. al, 2005), patient monitoring (Herbert, O'Donoghue, Ling, Fei & Fok, 2006) and robot navigation (Fok, Roman & Lu, 2006).

Since WSN nodes have limited hardware capacities, they need a special operating system. TinyOS is the most commonly used WSN operating system which is developed at UC Berkeley (Levis et al., 2004). As stated in TinyOS website, over 500 research groups and companies are using TinyOS in their research (http://www.tinyos.net/special/mission).

TinyOS has an embedded simulator called TOSSIM (Levis & Lee, 2003), which is a controlled and repeatable simulator that runs on a PC. A user can simulate WSN applications by compiling the application into the TOSSIM instead of real motes. TOSSIM does not have a graphical interface but TinyViz software can be used to visualize and control TOSSIM (Levis, Lee, Welsh & Culler, 2003).

Simulations have some advantages over real experiments. They are low-cost, visual, fast, scalable and configurable experiments. Hence, simulations play an important role in academia and industry as a necessary research method to analyze and justify theoretical models. However, there has not been any remarkable attempt to simulate mobile agents for WSNs.

Although TOSSIM and TinyViz are powerful tools for simulation of WSN applications, mobile agent middleware such as Agilla has not been transferred to the simulation environment. There is a necessity to develop software to fill this gap.

1.2. PURPOSE OF THE STUDY

The starting point of this study was to investigate existing middleware for wireless sensor networks and identify successful ones. In order to achieve this goal, the important criteria, which must be satisfied by a complete middleware are stated.

Middleware applications are then classified according to their programming approaches and noteworthy middleware of each approach is investigated. Advantages and disadvantages of these approaches and middleware are specified following the results of the survey. Eventually, agent based approach is determined to be the most advantageous approach and Agilla middleware is determined to be the most advantageous middleware. Some improvements are proposed to come out with a more efficient and more stable Agilla.

The second and more important purpose of this study was developing software to simulate mobile agents to fill this gap. The first effort is made to simulate mobile agents using an application. Agilla Simulator application was developed to transfer mobile agents to the simulation environment. Mobile agents can be composed and simulated via this Agilla Simulator, which also records log files during simulation.

The third purpose was to make simulations of mobile agents in order to achieve two important goals. One of them is benchmarking of mobile agent instructions to develop more efficient mobile agents. The other and more important goal is to validate simulated systems by comparison of results obtained from a simulated system with those of a real system. The result of simulations shows that the simulated system produced almost the same results of the real system. This result validates the simulated system and shows that simulation environment is useful to make successful simulations of real life situations.

1.3. OUTLINE OF THE THESIS

In Chapter 1, first the concept of middleware for wireless sensor networks and simulations of mobile agents are introduced. Then, significance of the study is briefly explained.

Wireless sensor networks and middleware are explained in chapter 2. This section first explains wireless sensor networks and their challenges. Next, hardware platforms that are used in WSNs are presented. Afterwards, middleware for WSNs and their challenges are stated. Furthermore, middleware approaches are presented by giving information about the major middleware for each approach including their advantages and disadvantages. Finally, advantages of mobile agent middleware are listed.

In Chapter 3, software which are used in this study are presented. These software are TinyOS, TOSSIM, TinyViz, Serial Forwarder and Agilla.

Improvements on Agilla are described in chapter 4. First, the suggested simulation application - Agilla Simulator - is presented. Later, bug fixes and added instructions to Agilla are explained.

In Chapter 5, simulation studies are presented. Advantages of simulations are explained first. Later, steps of setting up the simulation environment are presented. Then, simulations which are made to measure performance of important Agilla instructions are shown. Moreover, comparison of measurements obtained using the simulated system and reported experiments using real systems are presented. Lastly, results of simulation studies are discussed.

Finally, in chapter 6, summary of the work done and contributions of the thesis are presented. Further research opportunities are also suggested.

CHAPTER 2

WIRELESS SENSOR NETWORKS AND MIDDLEWARE

This chapter deals with the Wireless Sensor Networks (WSNs) and middleware for WSNs. Main characteristics of a WSN, challenges and current research areas on WSNs and notable WSN hardware platforms are presented in detail. Challenges of middleware development and middleware approaches are also mentioned. Moreover, some important characteristics of each middleware approach are summarized.

2.1. WIRELESS SENSOR NETWORKS

A Wireless Sensor Network (WSN) is composed of a large set of sensor nodes. Each sensor node has various sensors depending on the WSN application. Sensor nodes have storage and communication capability in addition to sensing capability. In general, sensor nodes collect data from the environment and communicate with each other using a multi hop approach. Then, a basestation receives the collected data. The basestation is connected to a laptop, PC or PDA as seen in Figure 2.1.

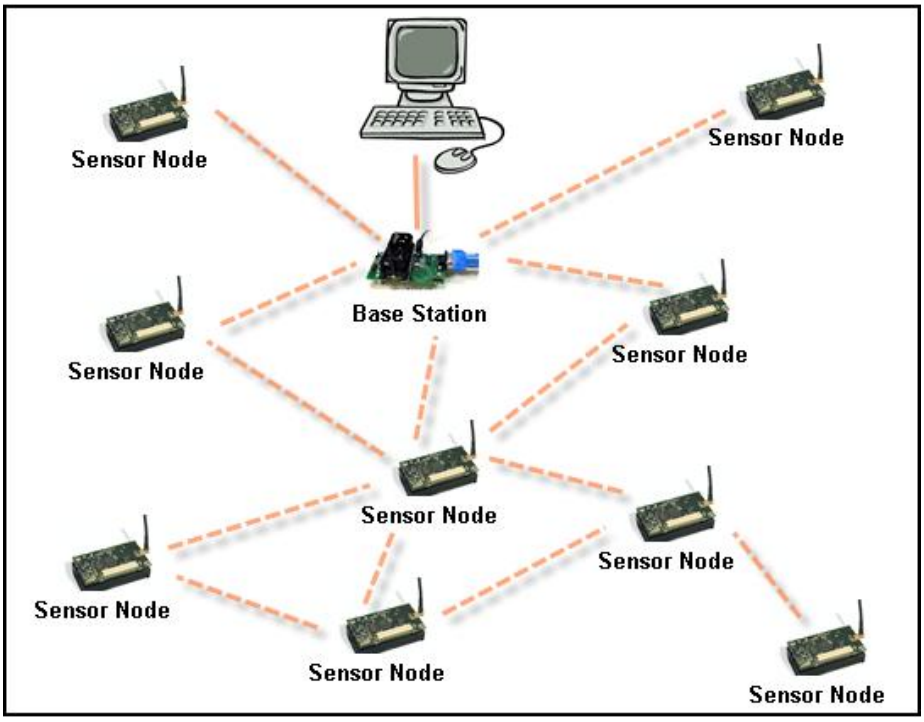

Figure 2.1: A Wireless Sensor Network

2.1.1. WSN Challenges and Current Research Areas

A sensor node has limited resources. Most important ones are:

- **Bandwidth:** The bandwidth of wireless links among sensor nodes is very limited. This communication constraint increases packet losses, latency and introduces communication failures (Ganesan, Krishnamachari, Woo, Culler, Estrin & Wicker, 2003).

- **Processing:** Sensor nodes have limited computing power; hence data processing power of a sensor node is limited.

- **Storage:** Because of limited memory of a senor node, its storage capability is restricted.

- **Power:** Sensor nodes have limited power supply, a typical sensor node works one week under full-load and one year in the idle state (Yao and Gehrke, 2002).

In addition to the above resource constraints, WSNs pose many other challenges, one of which is large scale of network deployment. Heterogeneity of nodes is also an important issue for WSNs. Another challenge is network longevity and robustness. Furthermore, adaptation to environmental changes is also essential for WSNs (Yu, Niyogi, Mehrotra & Venkatasubramanian, 2003). One other important challenge is programming the WSNs.

The WSN concept is an emerging research area with continuing work in various fields but most noteworthy ones are: routing (Shin, Song, Kim, Yu & Mah, 2007), security (Luk, Mezzour, Perrig & Gligor, 2007), radio management (Gao, Blow, Holding, Marshall & Peng, 2006) and middleware (Souto et al., 2006).

2.2. HARDWARE PLATFORMS

A WSN consists of small nodes called *motes*. Each mote has a processor, a radio unit and a sensor unit. Each sensor unit has different sensors, such as temperature sensor, humidity sensor and light sensor. Different mote types are used in WSN research. This section describes hardware architecture of the most used motes.

2.2.1. Mica Family Motes

Mica family sensor motes (Mica, Mica2, Mica2dot and MicaZ) are developed by UC Berkeley research group.

The first Mica family mote is called *Mica* mote (Hill and Culler, 2002). It consists of a sensor board (MTS board) and a mote processor/radio board, also known as MPR board. Figure 2.2. shows the configuration of the sensor board and the processor board. The MPR board consists of Atmel Atmega 128L processor, 916 MHz or 433 MHz RFM TR1000 radio transceiver and 2 AA batteries. TinyOS (Levis et al., 2004) operating system is embedded to Mica mote which will be discussed in Section 3.1.

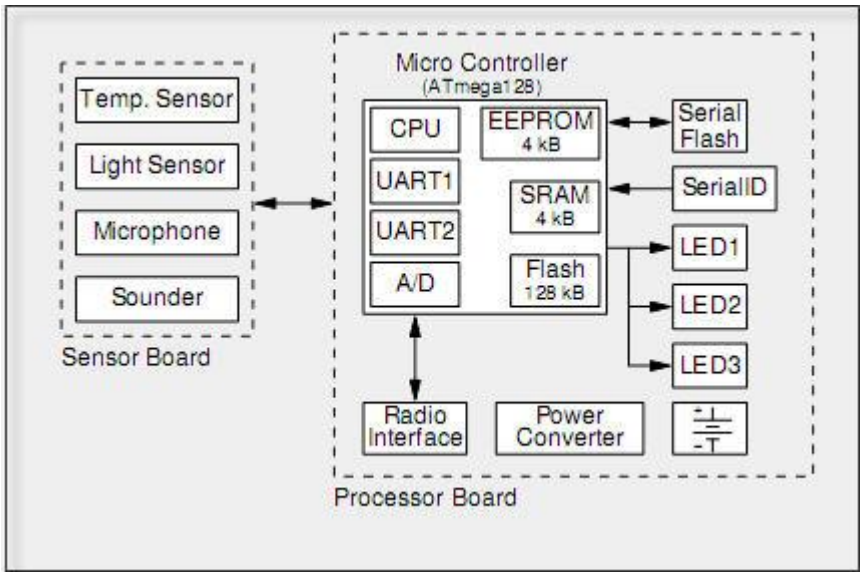

Figure 2.2: Diagram of Mica Mote

Atmel 128L processor is a low power, 4 MHz, 8 bit microcontroller. It has 4 KB of SRAM, 4 KB of EEPROM and 128 KB of internal flash memory which runs TinyOS.

Mica uses an ISM band radio transceiver module for wireless communication. The radio does not support buffering, so each bit must be served by the processor in real time. The range of the radio is 1 to 90 m. (30 m. under regular conditions).

A small 51 pin connector links the MPR and MTS boards. The interface includes an 8 channel 10-bit A/D converter, 2 serial UART ports, an I^2C serial port and PWM.

MTS board of Mica mote has temperature, light, microphone and sounder sensor. Light sensor measures ambient light value with a light dependent resistor (LDR). Temperature sensor measures temperature with a negative temperature coefficient (NTC) thermistor. There is a condenser microphone and 3 KHz acoustic sounder.

Following the Mica mote, Mica2 was developed. There are considerable improvements compared to the previous version. Most important design change is the new radio. Mica2 mote has Chipcon 1000 radio which has 150 to 300 m. range. The Chipcon 100 has digitally programmable output power, built-in Manchester encoding, software programmable frequencies and better noise immunity. Moreover, Mica2 has 512 KB flash memory and standalone boot loader. Mica2 support wireless remote programming.

Mica2dot is the smaller version of Mica2. They have similar features, but I/O capabilities of Mica2dot are degraded. It has 18 pins and 6 analog inputs. Instead of 2 AA batteries, it has a 3V coin cell battery. Mica2dot has only 25 mm diameter and 6 mm height. Mica and Mica2dot motes can be seen in Figure 2.3.

Last version of Mica family sensor motes is MicaZ mote. It is based on the IEEE 802.15.4 standard (Gutierrez, Naeve, Callaway, Bourgeois, Mitter & Heile, 2001). MicaZ has Chipcon 2420 radio.

Figure 2.3: Mica and Mica2dot Motes

2.2.2. MoteIV Motes

MoteIV Corporation started to design a mote with UC Berkeley research group in 2003. Their purpose was to develop an easy to use, standards based and power aware mote. They achieved their aims with the Telos mote which was released in 2004 (Polastre, Szewczyk & Culler, 2005).

Telos motes have two versions, Telos Revision A and Telos Revision B. Revision A has 8 MHz Texas Instruments MSP430 F149 microcontroller with 2KB of SRAM and 60KB of Flash, while revision B has 8 MHz Texas Instruments MSP430 F1611 microcontroller with 10KB of SRAM and 48KB of Flash.

Telos motes are IEEE 802.15.4 standard based (Gutierrez et al., 2001) and have the Chipcon CC2420 radio for wireless communications like the MicaZ mote. Humidity, temperature, and light sensors are integrated to Telos motes. They also support TinyOS like Mica family motes. On the other hand, unlike Mica family motes, they also have an USB connector.

After Telos motes, MoteIV developed Tmote Sky mote that used Telos as a reference design. However, Tmote Sky does not have considerable changes over Telos Revision B. Tmote Sky mote can be seen in Figure 2.4.

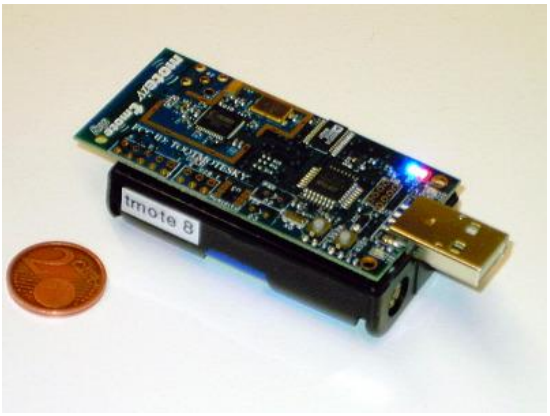

Figure 2.4: Tmote Sky mote

2.3. MIDDLEWARE FOR WIRELESS SENSOR NETWORKS

Unique characteristics and challenges of sensor nodes complicate application development for WSNs. Middleware layer of a WSN is between low level layers, such as hardware and operating system, and the application layer as seen in Figure 2.5. Middleware supports high-level abstractions to improve application development.

Recently, number of research projects on middleware development for WSNs have increased dramatically.

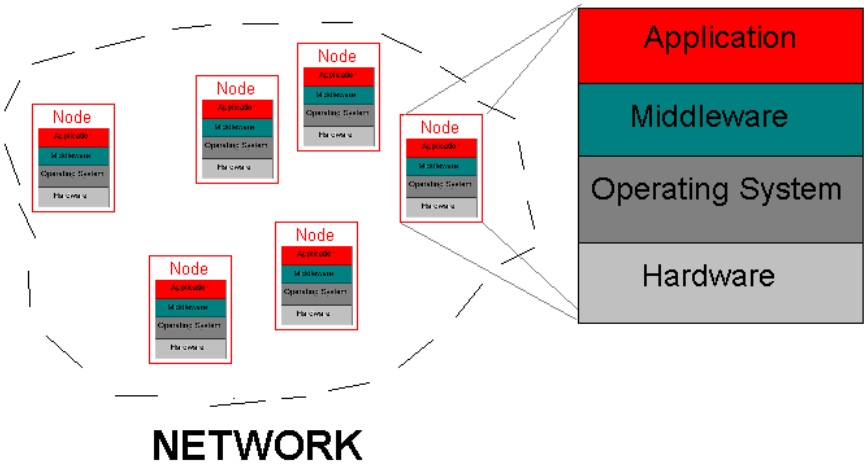

Figure 2.5: Common Architecture of Middleware

2.3.1. Middleware Challenges

There are numerous challenges imposed on middleware by WSN characteristics and applications. A successful WSN middleware must offer solutions to these challenges.

- **Heterogeneity:** A WSN may consist of various types of sensor nodes (Römer, Kastern & Mattern, 2002). Different nodes may have different hardware platforms. Moreover, a variety of applications can be implemented on a WSN. Since applications and hardware platforms are not homogeneous, a WSN middleware must support different types of hardware and applications.

- **Managing Resources:** Considering the limited resources of a WSN, a middleware must efficiently use the energy, memory, processor and bandwidth resources. A middleware should provide mechanisms to reduce power consumption while using memory and processor resources without waste (Hadim & Mohamed, 2006a). For example, a WSN middleware should support sleeping of nodes while they are idle.

- **Adaptability:** A WSN middleware should adapt to dynamic topology, dynamic environment and mobile nodes without affecting performance of the WSN. It should support fault tolerance and self-maintenance of sensor nodes (Hadim & Mohamed, 2006b).

- **Ease of Use:** Generally, WSN applications are complex programs. A middleware should provide easy programming and re-programming of WSNs. If a middleware's abstraction level is high, this middleware supposed to be easy to use. A middleware should hide the complexity of the WSN from the application developer (Römer, 2004).

- **Scalability:** A WSN can consist of hundreds to thousands of nodes. Furthermore, it can be used by multiple users. Consequently, a WSN middleware must support a large scale of nodes and multiple users. Moreover, a WSN middleware should provide mechanisms to allow expanding of an application without affecting the functionality of the WSN (Molla & Ahamed, 2006).

- **Security:** Although security plays an important role in WSN applications such as defense and healthcare applications, current middleware does not support security adequately. Wide deployment characteristic of WSNs increases their vulnerability level. A WSN is vulnerable to DoS attacks, malicious intruders, eavesdropping, traffic analyzes, node tampering and capture (Roman, Zhou and Lopez, 2005). Since WSNs have limited resources unlike traditional networks, conventional security methods are not suitable for them. Moreover, a WSN is often deployed in unsecured areas, and hence open to physical attacks. According to these issues, special solutions should be used by middleware to provide security.

- **Quality of Service (QoS):** QoS metrics (network coverage, network throughput, number of active nodes, data delivery delay, etc.) are important for all networks; however, current middleware usually do not provide quality of service aspects.

- Since there are some trade-offs between QoS parameters, a middleware should provide optimization of QoS (Yu, Krishnamachari & Prasanna, 2004).

2.4. MIDDLEWARE APPROACHES

WSN middleware can be classified with different perspectives. In this study, they are classified according to their programming approaches. Four main categories are defined: Database approach, virtual machine approach, adaptive approach and agent based approach. Most of the current middleware can fit into one of these categories but alternative categories can be suggested.

Comparison of middleware approaches considering previously mentioned challenges is shown in Table 2.1. These different approaches are discussed in detail in the following subsections.

Table 2.1: Comparison of Middleware Approaches

	Database Approach	**Virtual Machine Approach**	**Adaptive Approach**	**Agent Based Approach**
Heterogeneity	No	Some	No	Yes
Managing Resources	Yes	Yes	Yes	Yes
Adaptability	Some	Yes	Yes	Yes
Scalability	Some	Yes	Some	Yes
Ease of use	Yes	Some	Yes	Yes
Security	No	No	No	No
QoS	No	No	Yes	No

2.4.1. Database Approach

Cougar (Yao and Gehrke, 2002), DSWare (Yu et al., 2003), TinyDB (Madden, Franklin, Hellerstein & Hong, 2005) and SINA (Shen, Srisathapornphat & Jaikaeo, 2001) are members of this category. In the database approach, middleware abstracts the WSN as a virtual database and abstractions are focused on data rather than communication. Some SQL-like languages are used to query the WSN. Queries are injected from a base station.

This type of middleware have easy-to-use interface. However, this approach is not suitable for real-time applications and only approximate results are provided from the middleware. Moreover; security, hardware heterogeneity and QoS are not regarded by the database approach.

2.4.1.1. Cougar

Cougar is an instance of the database approach, developed at Cornell University. It represents the WSN as a virtual database system. This system includes sensor database and sensor queries. Sensor data is generated via signal processing operations and is stored in a sensor database. Queries are sent to the WSN, and the data can be pulled when the queries are processed.

Although Cougar distributes queries between sensor nodes to reduce energy consumption, it consumes more resources from other approaches because it transfers a large amount of raw data from sensor nodes to the database server.

Theoretically, Cougar supports large scale WSNs; however, the central optimizer used by Cougar to maintain global knowledge of the WSN is not suitable for large scale WSNs because of the dynamic temperament of WSNs. Nodes mobility and hardware heterogeneity are other unsolved issues for Cougar.

2.4.1.2. DSWare

DSWare (Data Service Middleware) is also a database middleware. It is suitable for event detection applications. DSWare provides group-based decision making and reliable data-centric storage. These features make DSWare flexible.

DSWare uses an SQL-like language for event operations. Therefore, it has an easy to use interface like other database approach middleware. DSWare can't fully support scalability, because the sensor database in each node requires continuous updating in highly dynamic applications. Nodes mobility is partially supported by DSWare but like Cougar, heterogeneity is another unsolved problem. On the other hand, its power and bandwidth consumption are in an acceptable level.

2.4.1.3. TinyDB

TinyDB is a declarative query-processing system which runs on top of TinyOS. TinyDB is developed by the Telegraph group of UC Berkeley. It uses a database table. Columns of the table include some information such as node id and sensor type.

When a user issues a query, TinyDB sends the query to all nodes, even nodes for which information is not needed. As a result, scalability is partially supported by TinyDB.

It uses a spanning tree for routing. Root of the tree is a particular node, and parent nodes are aggregation points for their child nodes. The tree is bidirectional, the query is automatically routed to all nodes, each node processes the query and resulting data is sent back in reverse direction. Maintenance of the tree is managed by TinyDB; hence the application developer does not need to pay attention to node failures or routing problems.

TinyDB uses a decentralized approach to process a query. Each sensor node has its own query processor to preprocess and aggregate sensor data enquired by the query specification.

TinyDB is an easy to use middleware since it uses a SQL-like language. Also routing and aggregation issues do not affect the user. Its GUI can be seen in Figure 2.6. Nevertheless, it isn't suitable for all applications since only predefined events are supported by TinyDB.

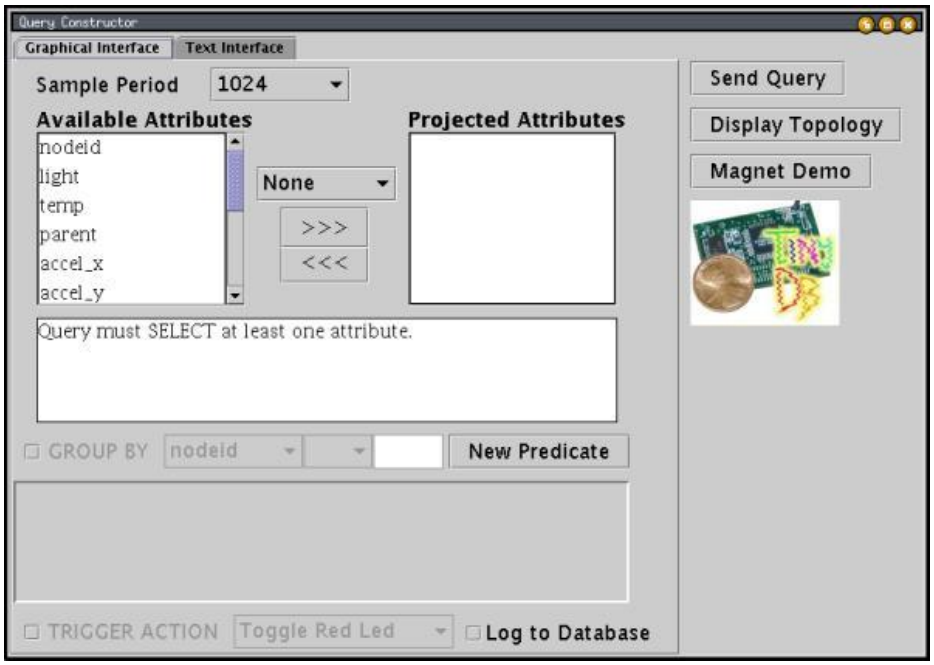

Figure 2.6: TinyDB GUI

2.4.1.4. SINA

SINA (Sensor Information Networking Architecture) is another database approach middleware that uses a query language. It was developed at the University of Delaware.

SINA represents the WSN as a spreadsheet database. Each database contains cells and each cell includes an attribute of a sensor node such as location and sensor reading. Attribute based naming makes it easy to sensor data.

SINA uses hierarchical clustering of sensors to provide efficient data aggregation. It also uses special protocols to restrict the rebroadcast of similar information to nodes.

SINA figures out mobility and power consumption issues with the aid of previously explained features. It is more flexible than other database approach middleware since it does not support only SQL-like languages, but also supports a scripting language called SQTL (Sensor Query and Tasking Language). This language provides basic operations for sensor hardware access, and event handling.

However, it does not support adaptability, since it does not provide mechanisms to handle frequently encountered WSN faults such as node failure. Thus, SINA isn't suitable for real-time applications. It supports scalability partially because of the fixed global network structure maintained by SINA. Moreover, hardware heterogeneity is still an unsolved problem.

2.4.2. Virtual Machine Approach

Maté (Levis & Culler, 2002) and Magnet (Barr et al., 2002) are considerable instances of this approach**.** Virtual machine middleware are based on code interpreters. They were implemented on top of operating systems. An application consists of small modules and runs on top of the middleware. Middleware inserts these modules into the WSN, and then the virtual machine interprets them.

The virtual machine system reduces energy consumption and network usage, so this approach can efficiently manage resources.

2.4.2.1. Maté

Maté middleware is a representative of a virtual machine approach developed at the University of California at Berkeley. It is a byte code interpreter implemented on TinyOS operating system. It has a specific scripting language called TinyScript. Maté provides little executable codes called "bytecode" to program WSNs. A bytecode program is a very compact program which is written in TinyScript.

Bytecode programs are divided into equal sized chunks capsules of 24 instructions where each instruction is a single byte long. This property provides easy injection of large programs into the WSN by allowing these programs to be made up of multiple small capsules.

If a bytecode program is injected to a WSN, it spreads through into the network until all nodes of the WSN have a clone of the program. Bytecode programs provide flexibility of Maté because they can move an update through the WSN easily. Maté is considered to be the origin of almost all agent based programs because of this feature.

One of the biggest disadvantages of Maté is that only one bytecode program can run on the WSN at once. This limitation decreases the flexibility of Maté.

Maté has five key components. These are the VM (Maté), the network, the logger, the hardware, and the boot/scheduler. Maté uses a synchronous event model. In reaction to an event, it starts execution. Therefore, Maté keeps away from message buffering and large storage. This model decreases bugs and increases simplicity of application level programming.

Maté has its own instruction set. A Maté program consists of low and high level instructions in a stack-based architecture. Arithmetic operations, loop operations and WSN-specific operations such as forwarding commands are some of the Maté instructions which allow sensor readings, messages and values as operands for these instructions. Maté's instruction set provides high level abstraction for an application developer.

Although Maté's programming model is simple and easy to use for the application developer, it is not flexible enough. This inflexibility inhibits Maté from covering a large spectrum of applications.

Magnet

Magnet is another middleware of the virtual machine approach developed at Cornell University. It has a network abstraction defined as Single System Image (SSI) which represents the whole network as a single Java Virtual Machine (JVM). The JVM system consists of a dynamic and a static component. The static component is accountable for rewriting regular Java applications designed for a single JVM. Also, this component injects these applications into the WSN. The dynamic component, placed on each WSN node, is responsible for monitoring applications and performing application specific tasks such as object creation and migration.

Java implementation of Magnet and SSI technique simplifies application development. Magnet provides power-aware policies such as NetPull and NetCenter. These policies reduce energy consumption. It is a general-purpose middleware which supports large scale applications. However, only partial heterogeneity is supported by Magnet because of JVM. Moreover, the JVM presents an overhead on Magnet's instructions.

2.4.3. Adaptive Approach

MiLAN (Murphy & Heinzelman, 2002) and AutoSec (Han & Venkatasubramanian, 2001) are some representatives of this approach. Adaptability is the main characteristic of this type of middleware. Applications can tune the network to meet their requirements with a QoS advantage. However, this approach is not suitable for general purpose use, because middleware is tightly coupled with applications.

2.4.3.1.1. MiLAN

MiLAN (Middleware Linking Applications and Networks) is an easy-to-use adaptive approach middleware developed at the University of Rochester. MiLAN acts a layer on top of multiple physical networks. Through this feature, MiLAN applications can specify their QoS requirements and tune network characteristics

to achieve their goals. MiLAN uses graph theory and service-discovery protocols such as Service Discovery Protocol and Service Location Protocol to determine which nodes are accessible or not. Nevertheless, this solution is centralized.

MiLAN middleware is tightly coupled with the application; therefore, it does not support hardware heterogeneity. Also, MiLAN does not provide mobility. In spite of these disadvantages, MiLAN is a scalable and power aware middleware.

2.4.3.2. AutoSeC

AutoSeC (Automatic Service Composition) is an adaptive approach middleware developed at UC Irvine. It is a dynamic service broker framework that provides effective utilization of system resources. AutoSeC provides access control for applications to provide QoS requirements on per-sensor basis. It is a power aware middleware but it does not support mobility and heterogeneity.

2.4.4. Agent Based Approach

Agilla (Fok et al., 2005) and Smart Messages (Kang, Borcea, Xu, Saxena, Kremer & Iftode, 2004) are examples of this type of middleware. Modular applications called “mobile agents” are used in the agent based approach. Mobile agents can be easily injected into the WSN. After injection, they can distribute themselves through the network.

2.4.4.1. Smart Messages

Smart Messages is a mobile agent middleware. The term “Smart Message” (SM) term is used for a user-defined distributed program which can execute on nodes and migrate between nodes to reach other nodes. If a SM is required to migrate between two nodes and there are some other nodes between these two nodes, it can self-root. Codes of the SM and execution state are carried during migration. Each node has its virtual machine to execute the SM. A name based memory called “tag space” is used as shared memory between SMs.

2.4.4.2. Agilla

Agilla is another mobile agent middleware. Mobile agents are dynamic and intelligent programs that can migrate through nodes. Each mobile agent has its own code and execution state. After its injection, the mobile agent performs autonomously. When the agent reaches a sensor node, it runs its instructions. To achieve its goal, a mobile agent can move or clone itself to another sensor node or interplay with other agents.

Most significant features of mobile agents are:

- Agents can move or clone from one node to another node (migration).
- Each agent works as a virtual machine which data memory and dedicated instructions which they can execute.
- Multiple agents can exist on a node at the same time.

Agilla and Smart Messages have common characteristics. Both of them use mobile agents (only their names are different), support migration and use a local shared memory (tuple space in Agilla and tag space in Smart Messages) to provide local communication.

However, Agilla has several advantages over Smart Messages. A node can run only a single execution thread in Smart Messages. On the other hand, multiple mobile agents can exist on a node simultaneously in Agilla. In other words, Agilla supports coexistence of multiple applications on a single node. Moreover, Smart Messages does not support inter-node communication while Agilla does. Each smart message is responsible for its own routing in Smart Messages, and the user must implement a routing algorithm in this middleware. However, Agilla middleware is responsible for routing mobile agents; users are not responsible for routing. Also, Agilla is easier to use then Smart Messages.

Agilla will be discussed in Section 3.5. in more detail.

2.5. ADVANTAGES OF MOBILE AGENT MIDDLEWARE

Mobile agents have been used in networks for years as an alternative to traditional client-server models and their advantages are well known (Lange & Ohima, 1999, Chess, Grosof, Harrison, Levine, Parris &Tsudik, 1995). Some examples of old mobile agent applications for networking are TACOMA (Johansen, Renesse & Scheneider, 1995), MARS (Cabri, Leonardi & Zambonelli, 2000) and PeerWare (Picco & Cugola, 2001).

Mobile agent middleware have several advantages over the other approaches. These are:

- Adaptation to environmental changes and wireless reprogramming are two important challenges for WSNs. Assume a WSN is primarily deployed for intrusion detection in a building. Civil defense authorities may want to reprogram the network to detect fire or gas leak in an emergency situation. Installing all these applications at once is not flexible, manageable or scalable. Mobile agent middleware addresses this problem. It provides dynamic reprogramming of WSNs by allowing new agents to be injected where old agents die. Mobile agent middleware support adaptability and mobility.

- Since multiple agents can exist on a node simultaneously, mobile agent middleware support the coexistence of multiple applications on a node. As an example, Fok et al. (2005) used two mobile agents for a fire-tracking application.

- Shared memory model of the tuple spaces/tag spaces enables one agent to insert a tuple which contains a data and another agent to retrieve this data later. This feature allows coordination of agents to perform a common task in a highly decoupled fashion. This model provides scalability of the middleware.

- Mobile agents efficiently use resources of sensor nodes as they only need resources for visited nodes. Mobile agent middleware are power aware.
- WSN application development with mobile agents is easier than development with other approaches.

CHAPTER 3

SOFTWARE ARCHITECTURE

This chapter presents the software architecture used in this study. First, TinyOS operating system is discussed. Then, TOSSIM simulator and TinyViz visualization tool are mentioned. Finally, Agilla middleware is explained in detail.

3.1. TINYOS

Existing operating systems such as Linux, Windows and MacOS were designed for PC-like complex systems. However, Wireless Sensor Networks (WSNs) have limited hardware capabilities from the point of power, processor and memory capacities. A WSN operating system must satisfy these strict application requirements. Moreover, it must allow multiple applications to simultaneously use system resources such as computation, communication and memory. Existing embedded device operating systems do not meet these requirements. Therefore, there was a necessity to develop a WSN–specific operating system. Such a WSN operating system was built at the UC Berkeley, and named TinyOS (Levis et al., 2004).

TinyOS is the most common WSN operating system. As stated in TinyOS website, over 500 research groups and companies are using TinyOS for their research (http://www.tinyos.net/special/mission). TinyOS is an event driven, light-weight, open source and modular operating system that is very well-suited

for WSNs. TinyOS can manage the bounded hardware capabilities effectively. Abstraction layers were used in TinyOS to support different hardware platforms.

TinyOS architecture is component based. Since user applications can include only necessary components, this feature minimizes code size.

TinyOS was written in NesC, a dialect of C language (Gay, Levis, Behren, Welsh, Brewer & Culler, 2003). NesC features an event driven execution model. This property is important, because WSNs usually operate on event-driven basis.

TinyOS achieves microthreading by means of two main threads: tasks and hardware events. TinyOS utilizes event based execution to provide high levels of operating efficiency required in wireless sensor networks. Event based execution model allows for high levels of concurrency to be treated in a very small amount of space.

TinyOS holds a two-level scheduling structure. This structure provides immediate performing of events while long running tasks are interrupted. Tasks are run to completion and cannot preempt each other but events can preempt other events and tasks. Tasks can be used for non time-critical computation processes, and events can be used for operations where timing requirements are strict. This execution model is similar to finite state machine models.

3.2. TOSSIM

TOSSIM is a scalable simulation framework for WSNs that run TinyOS (Levis & Lee, 2003). TOSSIM is part of the TinyOS operating system. It is completely integrated with TinyOS and NesC. Users can test a real TinyOS application by compiling the application into the TOSSIM instead of real motes. TOSSIM is a controlled and repeatable environment which runs on a PC.

Figure 3.1 shows that TOSSIM has component based architecture like TinyOS. This feature makes TOSSIM a very efficient platform. In the simulation of a node, TOSSIM executes only hardware interface components (simple radio stack,

sensors and other peripherals) while other components run unaltered on top of these. For simulation of the whole WSN, TOSSIM creates separate instances of the components graph for each node. A discrete event queue simulates hardware interactions.

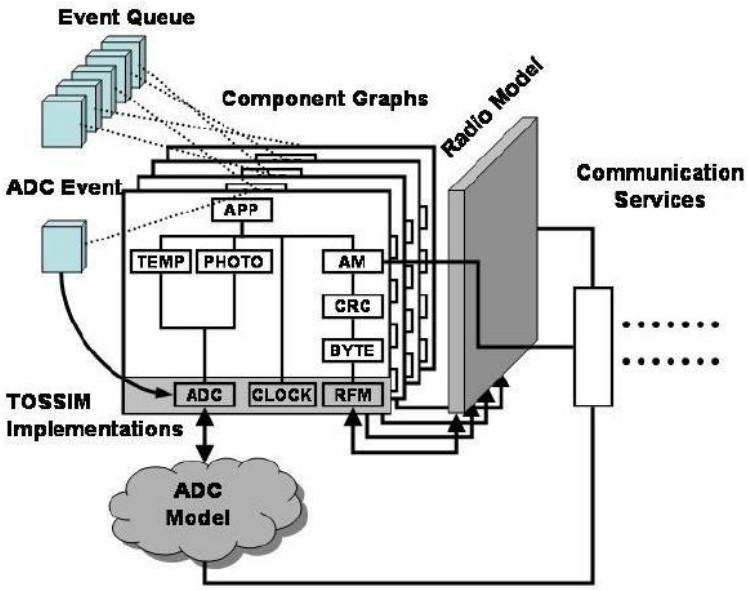

Figure 3.1: TOSSIM Architecture (Levis, Lee, Welsh & Culler, 2003)

TOSSIM is a bit level discrete event simulator. It simulates every analog-digital converter (ADC) capture and every interrupt in the system. It compiles a TinyOS application into a native executable form that runs on a PC. TOSSIM can support thousands of nodes through this feature.

Bit level simulation is used for the simulation of radio communication between nodes. Various radio propagation models are supported by TOSSIM.

TOSSIM supports communication of the simulation results with external programs. This feature provides controlling, visualization and debugging of simulation by external tools, such as TinyViz (Levis et al., 2003).

TOSSIM simulator has some characteristics which differentiate it from other simulators:

- **Completeness:** A TOSSIM simulation can handle almost all the system interactions.
- **Fidelity**: The simulator can capture the detailed behavior of the nodes (ADC captures, interrupts etc.).
- **Scalability:** Since TOSSIM has the capability to simulate a large numbers of nodes simultaneously, it can simulate an entire network.
- **Bridging:** TOSSIM bridges the gap between algorithm and implementation. Since TOSSIM uses the same code that is used to program the hardware, it can detect errors in implementation.

Although TOSSIM can simulate real world events, it has some limitations:

- TOSSIM cannot simulate a heterogeneous network of sensors. All sensors must be the same type.
- All motes in a TOSSIM simulator must contain the same application.
- TOSSIM does not model execution time (Levis et al., 2003). Time is fixed at 4 MHz granularity (CPU clock rate of Mica) and unchangeable.
- TOSSIM does not model radio propagation model by itself; rather, it provides radio abstraction and an external program like TinyViz provides the intended radio model.
- TOSSIM does not model energy consumption. External programs can be used to calculate this.

3.3. TINYVIZ

TinyViz is a visualization, actuation and control tool written in Java for TOSSIM (Levis et al., 2003). TinyViz can capture all of the events in a simulation and visualize sensor readings, leds and radio links. TinyViz allows simulations to be visualized, analyzed and controlled. Its GUI can be seen in Figure 3.2.

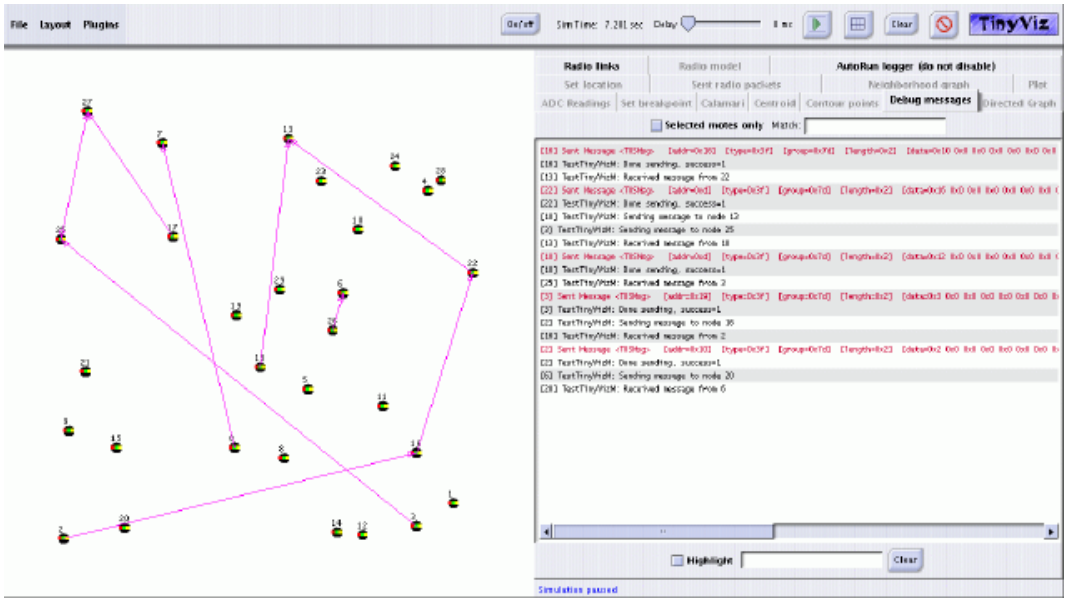

Figure 3.2: TinyViz GUI

Since TinyOS has an event-based execution model, the core TinyViz engine uses an event-driven model too. This model provides mapping of TinyOS events and TinyViz. TinyViz engine uses an event bus to read events from simulation. It read events and notifies all active plugins about events.

By itself, TinyViz can draw motes and their leds. However, various plugins could be developed for TinyViz which can do the intended tasks, such as visualizing network traffic. Plugins provide user-simulation interaction. For example, plugins can send commands to TOSSIM which can be enabled or disabled during simulation.

TinyViz has a default built-in plugin set. These plugins provide basic debugging and analyzing functions. In addition, developers can implement their own application-specific plugins.

The basic built-in plugins are:

- **ADC Readings plugin:** Users can observe the ADC channel via this plugin. ADC Channel converts the output voltage into an integer.

- **Debug Messages Plugin:** During simulation, TOSSIM generates some debug messages about the simulation. Generated debug messages can be monitored by this plugin in a window. Moreover, user can filter debug messages by selecting intended group of motes. User can also enter a pattern to highlight matched debug messages.

- **Layout Plugin:** There are three layout options: random, grid-based and "grid + random" layout. First, the location of the motes on the GUI is used to determine radio connectivity, when the Radio Model Plugin is enabled. Second, it is used to set the virtual location of the motes, when the Location Plugin is enabled.

- **Location Plugin:** Location plugin shows the virtual location of each mote. This plugin can be used as a real localization service.

- **Radio Links Plugin:** This plugin visualizes radio message activity. When a mote broadcasts a message, a blue circle is drawn around it to represent that the mote is broadcasting a message. Message transfer between two nodes is specified with a directed arrow.

- **Radio Model Plugin:** This plugin changes radio connectivity based on distances between motes in the GUI. Link probabilities are graphically displayed with this plugin.

- **Set Breakpoint Plugin:** This plugin pauses the simulation when some condition is realized. The possible conditions are a substring match on a debug message, or a match on the contents of a sent radio message. Multiple breakpoints can be set. Breakpoints can be enabled or disabled in the breakpoints list.

- **Sent Radio Packets Plugin:** Although debug messages plugin also shows this information, this plugin shows all sent radio packets.

3.4. SERIAL FORWARDER

Motes and external applications which run on a PC must be connected to each other. This connection is provided over a TCP/IP stream by the Serial Forwarder application of TinyOS as shown in Figure 3.3. It runs as a server on the host machine and forwards packets from external applications to motes and vice-versa. Serial forwarder connects to the serial port of a real mote; this property is the source of its name.

In the simulation environment, TOSSIM uses its serial input control to retrieve data from simulation rather than real motes.

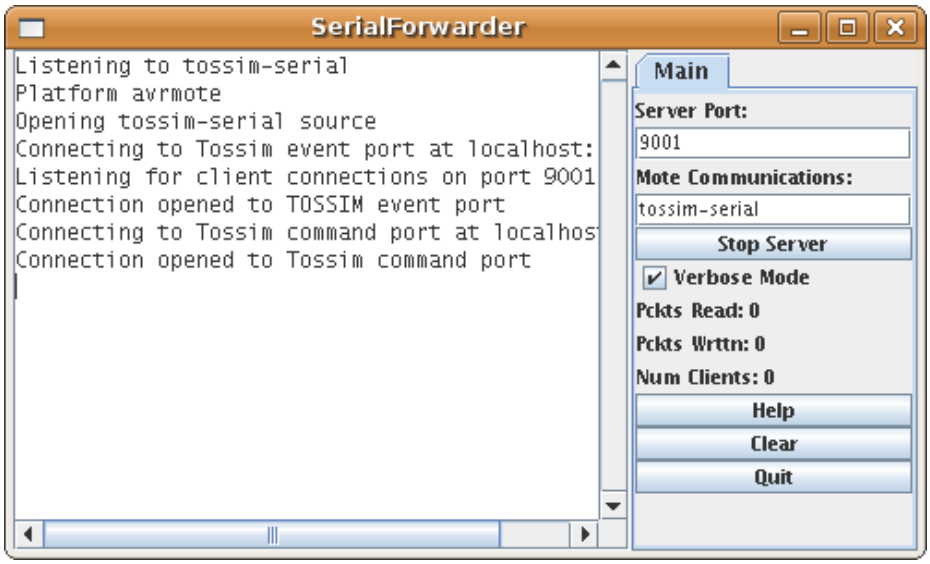

Figure 3.3: Serial Forwarder GUI

3.5. AGILLA

General characteristics of Agilla are mentioned in Section 2.4.4.2. This section provides a detailed look at Agilla.

3.5.1. Agilla Model

Agilla is initially developed for Mica2 motes which are described in Section 2.2.1. Initially, Agilla middleware must be loaded on the nodes before injecting mobile agents on them. After this operation, mobile agents can be injected to the WSN for performing their tasks.

Neighbor list, tuple space and migration terms are important concepts which comprise the Agilla model as can be seen in Figure 3.4.

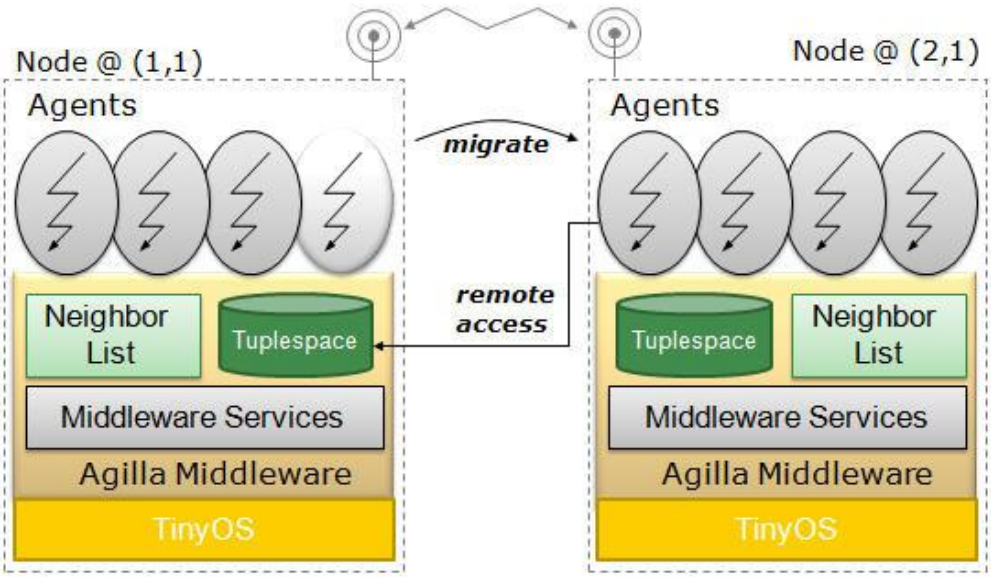

Figure 3.4: Agilla's System Architecture (Fok et al., 2006)

3.5.1.1. Tuple Space

WSN applications are often complex applications. A WSN application developed for Agilla can be composed of multiple autonomous agents. For instance, in a smart home application, there are a gas leak detection agent, an intruder detection

agent, an intruder tracker agent etc. To achieve the results expected from the application, Linda-like tuple space approach is used (Gelernter, 1985). Tuple space provides coordination of the agents.

In this approach, each node has a tuple space. Agents residing on the node (local agents) and agents that reside on other nodes (remote agents) can access tuple space of the same node. Therefore, each tuple space is shared by local and remote agents. Tuple space approach provides autonomy of agents and context discovery.

Each tuple in the tuple space is a sorted set of records and each record has a type (integer, string, location or sensor reading) and value.

An agent can access the tuple space with pattern matching. Patterns are called templates. A template is a sorted set of a record like a tuple. However, templates can contain wild cards.

A template matches a tuple if:

- They have same number of records
- Each record matches the same ordered record.

If there are numerous tuples which matches a template, the middleware chooses one of them non-deterministically. Even if the agent dies or moves, tuples remain in the tuple space. Only special instructions (in and rinp) can remove a tuple from the tuple space.

Agilla system has both local tuple space operations and remote tuple space operations. These operations are important in order to understand the tuple space concept.

Local tuple space operations are:

- **out:** This instruction inserts a tuple into the local tuple space. This tuple becomes shared by agents.

- **in:** It searches the node's tuple space for template matching. If a matching tuple is found, it is retrieved and removed from the tuple space. If there are no matching entries in the tuple space, this instruction blocks the tuple space until a match is found.

- **rd:** This instruction searches the node's tuple space for template matching like "in". However, it does not remove the matched tuple from the tuple space.

- **inp:** The difference with "in" and "inp" is "in" blocks the tuple space while "inp" does not.

- **rdp:** "rd" blocks tuple space while "rdp" does not.

Agilla uses remote tuple space operations to provide coordination of agents. Agents can remotely access tuple spaces on other nodes. Only non-blocking local operations (out, inp and rdp) have remote versions (rout, rinp and rrdp).

Remote tuple space operations are:

- **rout:** Remote version of "out".

- **rinp:** Remote version of "inp".

- **rrdp:** Remote version of "rdp".

- **routg:** This instruction inserts a tuple into tuple spaces of all one hop neighbor nodes. *routg* is the group version of the *rout* instruction.

- **rrdpg:** This instruction searches tuple spaces of all one hop neighbors for template matching. *rrdpg* is the group version of *rrdp* instruction.

Directions of some important tuple space operations are shown in Figure 3.5. to clarify some issues.

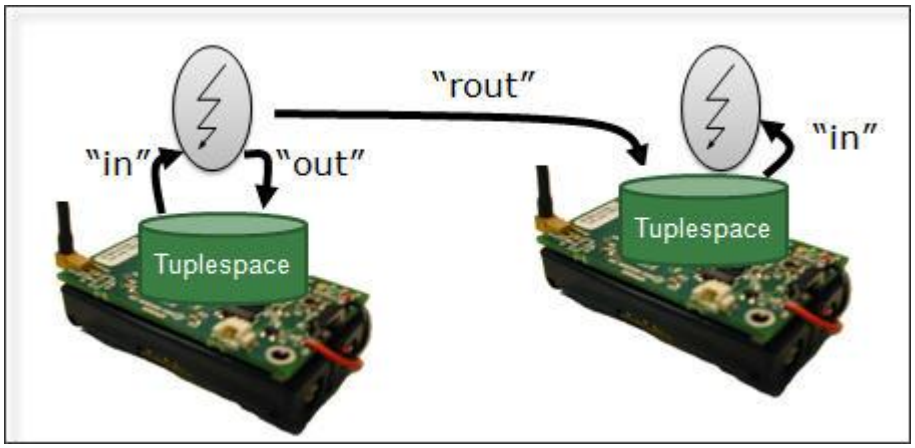

Figure 3.5: Tuple space operations

3.5.1.2. Neighbor List

Neighbor list consists of addresses of one hop neighbors. Agilla uses beacons for neighbor discovery. Each node has its own neighbor list and they are maintained by Agilla.

Agents can access the neighbor list using the following instructions:

- **numnbr**: This instruction retrieves number of neighbors in the list.
- **getnbr**: This instruction gets the address of the neighbor specified with a variable. This variable is the position of the neighbor in the list.
- **cisnbr**: This instruction checks if the specified location is a neighbor or not.
- **randnbr**: This instruction fetches the location of a random neighbor.

3.5.1.3. Migration

The term “migration” consists of moving and cloning of an agent. Moving of an agent is the transferring of the agent from one node to another node, whereas cloning is copying an agent from one node to another node.

Special instructions are used to move or clone one agent from one node to another. Cloning instructions are *sclone* and *wclone*. Moving instructions are *smove* and *wmove*.

First letters (s and w) indicate whether the migration is strong or weak. The difference is the transferred parts of agent. In strong migration, agent code, program counter, heap, stack and reactions are transferred and agent resumes running where it stopped. On the other hand, in a weak migration ,only the code is transferred and agent resumes execution from the beginning. There is some trade-offs of using strong or weak migration. Since strong migration transfers everything, programming is simplified but overhead is increased. An agent can migrate to any node even if it is multi-hop away. Middleware provides multi-hop migration transparently from user.

3.5.2. Implementation Platform of Agilla

Agilla was originally implemented on Mica2 motes. Now, Agilla can work on MicaZ, Tyndall 25mm and Tmote Sky motes. They were discussed in Section 2.2.2. TinyOS operating system runs on these motes and Agilla middleware must be loaded on TinyOS.

3.5.3. Agilla Architecture

Agilla's architecture has three layers as can be seen in Figure 3.6. : TinyOS, Agilla middleware, and mobile agents.

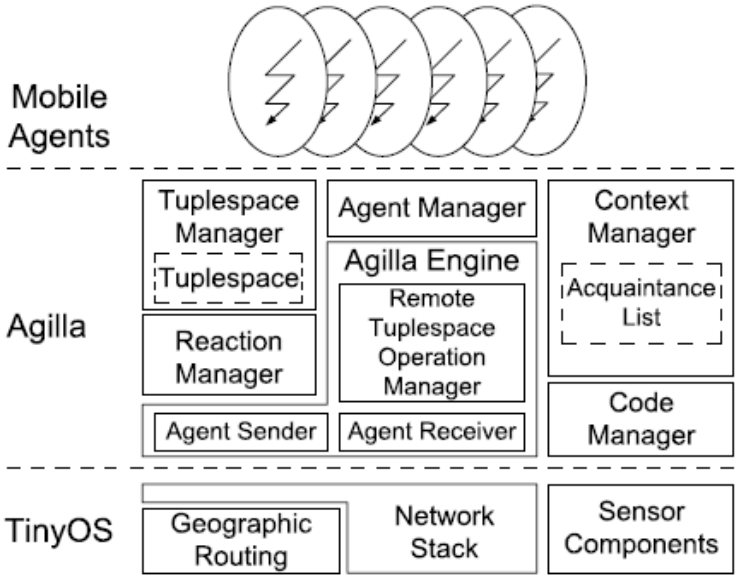

Figure 3.6: Agilla Architecture (Fok et al., 2006)

3.5.3.1. TinyOS

TinyOS is the lower layer of the architecture which was mentioned in Section 3.1. Geographic routing, network stack and sensor components are managed by TinyOS.

3.5.3.2. Mobile Agents

Mobile agents constitute the top layer of the architecture. They will be discussed in Section 3.5.5.

3.5.3.3. Agilla Middleware

Middle layer of the architecture is the core component of the Agilla middleware. Five managers together with the Agilla engine compose the middleware. Each of these is implemented as a TinyOS component and a separate process. These managers are tuple space manager, reaction manager, agent manager, context manager and code manager.

- **Tuple Space Manager:** Tuple space manager is responsible for non-blocking tuple space operations and local tuple space content. Blocking operations are implemented in the agent. Initially, 100 bytes is allocated for tuple space and each tuple is at most 18 bytes. Therefore, a tuple space can consist of at most 5 tuples.

- **Reaction Manager:** Reactions of agents are stored by the reaction manager in the reaction registry. Assume that an agent registers a reaction. In this case, the reaction manager searches a matched tuple in the tuple space and informs the agent manager if there is a matched tuple. Then, the agent manager pushes the tuple onto the operand stack of the agent and executes the reaction's code. Similarly, suppose an agent inserts a tuple into the tuple space. In this instance, the reaction manager looks for a matched reaction in the reaction registry. If a matched reaction with the inserted tuple is found, reaction manager fires the reaction.

  If an agent moves or clones to a node, reaction manager of the current node packages reactions of the agent into messages for transfer. If an agent arrives at a node, reaction manager adds reactions of this agent into the reaction registry.

  Initially, the reaction registry is 130 bytes. A reaction is divided into 22 bytes fixed sized blocks like code memory. Therefore, a reaction registry can consist of at most 5 reactions.

- **Agent Manager:** Context of an agent is maintained by the agent manager. If an agent arrives at a node, the agent manager allocates memory for this agent. After memory allocation, the agent manager informs the Agilla engine. Similarly, if an agent moves to another node or dies, the agent manager deallocates assigned memory for this agent.

  An agent's memory is composed of an execution state memory and a code memory. Code memory is divided into 22 byte blocks. When an agent

arrives at a node, the agent manager allocates the minimum number of blocks for it. For example, a migrating agent's code is 60 bytes. Therefore, the agent manager allocates $\lceil 60/22 \rceil = 3$ blocks for this agent. If there is not sufficient memory, migration of the agent will be aborted. Then, the agent continues running at its original node with the condition code set to 0. Execution state memory consists of heap, stack, condition code and program counter.

If an agent arrives at a node after a clone operation, the agent manager determines whether the agent is original or a clone.

- **Context Manager:** Context manager of a node is responsible for determining the location of the node and its neighbors. It maintains a neighbor list. Neighbor list operations were discussed in Section 3.5.1.2.

- **Code Manager:** Code manager is responsible for fetching the next instruction when agents are running as well as packaging the agent's code into messages. Each message is a 22 byte block. The reason for 22 bytes has to do with the TinyOS message format. A TinyOS message has 29-36 bytes payload. 22 bytes is guaranteed to fit within a TinyOS message.

- **Agilla Engine:** Agilla engine manages all other managers and runs as a virtual machine kernel. Agilla engine consists of an agent sender, an agent receiver and a remote tuple space operation manager.

  Agent sender and agent receiver are responsible for migration of agents. For migration of agents, one-hop communication is used. As mentioned above, agents are divided to message packets during migration. In one hop communication, an agent migrates one hop at a time and each message packet is acknowledged. Since Mica2's radio is highly unreliable, end-to-end communication was not used instead of hop-by-hop communication. Remote tuple space manager manages remote tuple space operations. These operations were discussed in Section 3.5.1.1.

3.5.4. Instruction Set Architecture (ISA)

Even though Agilla's ISA is based on Maté, there are many differences between tuple space and migration instructions while general purpose instructions are nearly identical to Mate's instructions. Tuple space instructions were discussed in Section 3.5.1.1 and migration instructions were discussed in Section 3.5.1.3.

3.5.5. Agent Architecture

A mobile agent is composed of a stack, heap, various registers and code as shown in Figure 3.7.

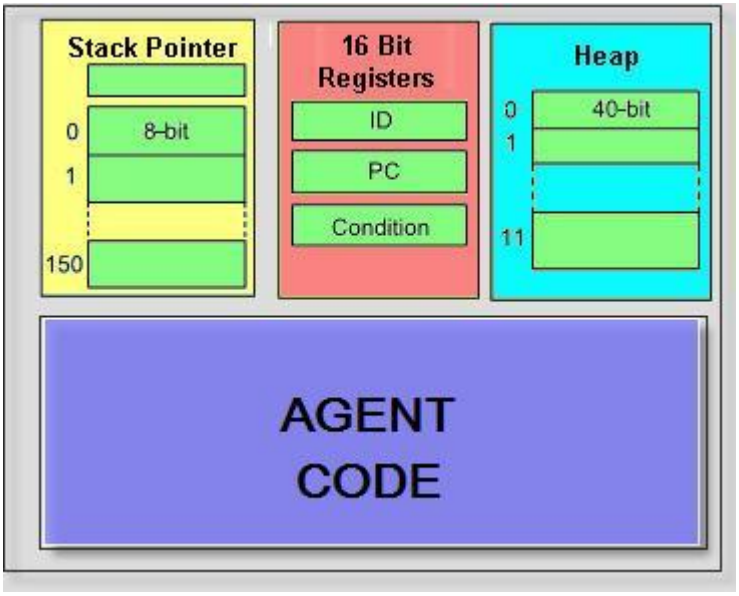

Figure 3.7: The Mobile Agent Architecture

- **Stack Pointer:** Stack based architecture of mobile agents allows small instructions. Most Agilla instructions are a single byte (Fok et al., 2005). Stack pointer indicates the current top of the stack.

- **Heap:** It is a random-access storage area. An agent can store 12 variables in its heap. *Setvar* instruction can store a variable to the heap and *getvar* instruction can get a variable from the heap.

- **16-bit Registers:** Unique ID of the agent, the program counter (PC), and the condition code are stored in the 16-bit registers. The unique ID of the agent is maintained across migration operations. A new ID is assigned to cloned agents. The PC is the address of the next instruction. It is maintained by the jump instructions. The code manager which is mentioned in Section 3.5.3.3. uses the PC for fetching the next instruction. The condition code records execution status. For example, the instruction *randnbr* sets the condition to be 1 if there is a neighbor node.

- **Code:** Code of an agent is composed of instructions. The code is divided to 22 byte blocks to fit within a TinyOS message, and an agent can have maximum of 15 blocks of code, so a code can be 330 bytes. One byte instructions mean that an agent can have up to 330 instructions.

CHAPTER 4

MODIFICATIONS ON AGILLA

This chapter presents our contributions to Agilla. The application for simulation of Agilla developed in this study will be presented first. This application will be included in newer versions of Agilla as offered by the main developer of Agilla, C. L. Fok. After presenting Agilla Simulator, important bug fixes will be explained. Later, new instructions which are added will be discussed.

4.1. AGILLA SIMULATOR

We developed a Java application to simulate Agilla which is called *Agilla Simulator*. Agilla agents can be simulated via this application.

Agilla Simulator simulates Agilla agents in 6 steps:

1. Opening or creating an agent
2. Converting the agent's code to NesC code
3. Embedding the converted code into the AgentMgrM.nc file
4. Compiling Agilla
5. Launching Serial Forwarder (SF)
6. Running the simulation with TinyViz and saving log files

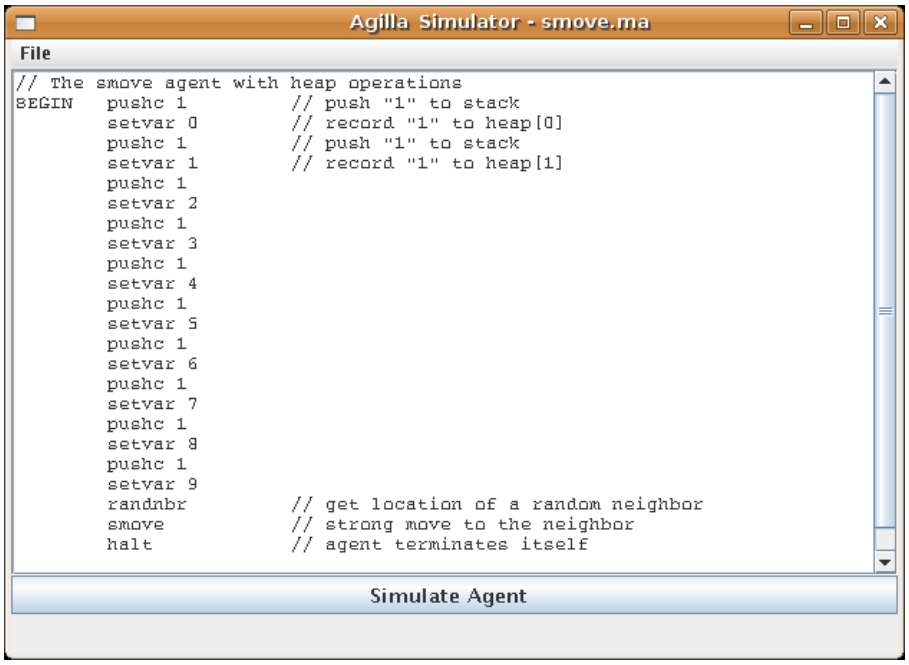

Figure 4.1: Agilla Simulator GUI

GUI of the Agilla Simulator is shown in Figure 4.1.

4.1.1. Opening or Creating an Agent

The user should open a previously saved agent or create a new agent. If the user wants to simulate an agent which was created before, he/she can open the agent's file. The user can also write a new agent in the workspace of the Agilla Simulator application. The created file can be saved to be used afterwards.

4.1.2. Converting the Agent's Code to NesC Code

As mentioned in Section 3.1., TinyOS was written in NesC programming language (Gay et al., 2003). NesC is designed especially for TinyOS. Programs for TinyOS are also written in NesC and this also applies to Agilla.

The code portion of an agent is composed of instructions of Agilla's ISA which is mentioned in Section 3.5.4. The code of the agent must be converted to NesC codes to be compiled and simulated. Agilla Simulator converts the agent's code to NesC line by line.

4.1.3. Embedding the Converted Code

After conversion of the code, Agilla Simulator application embeds the converted code of the agent into Agilla middleware. The application uses the AgentMgrM.nc file to simulate the agent. AgentMgrM.nc file is an Agilla component. Generally, it manages context of the agents.

AgentMgrM.nc file has several important functions:

- It migrates an agent from a host node to a destination node.
- Whenever a new agent arrives at a node, it allocates resources for this agent.
- If an agent dies or moves, it frees all resources (memory etc.) used by the agent.

AgentMgrM.nc file is used to simulate Agilla agents, because this component runs automatically when Agilla middleware is compiled on the network. Agilla Simulator inserts the intended agent into this file. After insertion of the NesC code of the agent, Agilla can be compiled for simulation of the agent.

4.1.4. Compiling Agilla

In real experiments, applications are compiled on the (real) wireless sensor network with "make *motename*" command. For example, if you want to compile your application on Mica2 motes, you should use "make mica2" command in your application folder.

On the other hand, you can compile your application on your PC with “make pc” command to simulate your application. After compilation of the application, TinyOS creates a “main.exe” file.

Agilla Simulator compiles Agilla to simulate the agent after embedding the NesC codes of the agent into Agilla.

4.1.5. Launching Serial Forwarder (SF) and Connecting TOSSIM to SF.

SF was mentioned in Section 3.4. After compiling Agilla, Agilla Simulator launches SF to make a connection to TOSSIM.

4.1.6. Running Simulation with TinyViz and Saving Log Files

Agilla Simulator uses TinyViz as a visual simulator. TinyViz is a visualization tool for TOSSIM simulator which was mentioned in Section 3.3.

A user can use TinyViz to simulate an application with “tinyviz -run build/pc/main.exe *number_of_nodes*” command in the application folder. For instance, if the user wants to simulate the application with 30 nodes, he/she should use “tinyviz -run build/pc/main.exe 30” command.

TinyViz is a powerful tool with an autorun feature which permits setting of the parameters automatically, running multiple simulations, logging data to files, taking screenshots and some other operations.

Various autorun files, shown in Appendix A., are used to specify properties of a TinyViz simulation, such as number of simulations, names of log files, number of simulation seconds elapsed, etc.

Agilla Simulator automatically runs TinyViz after connecting to SF without the interaction of the user. Moreover, it saves debug messages of the simulation to log files. We had approximately 2000 log files after simulations. Since manually analyzing these data is impossible, AWK programming language was used to analyze the log files.

4.1.7. Advantages Over the Simulation Method Described in the Agilla Website

A manual agent simulation method is mentioned on the Agilla website (http://mobilab.wustl.edu/projects/agilla/docs/tutorials/9_debug.html). Their method uses AgentMgrM.nc file like our method. However, there are major differences between the two methods:

- Their method does not use an application which has a GUI. All processes are manual.
- User has to use NesC code to program the agent in their method. However, the user does not have to know the NesC code in our proposed method.
- User has to manually open AgentMgrM.nc file and paste the NesC code into this file at the appropriate location in the method mentioned in the Agilla website. User may corrupt AgentMgrM.nc file in this case because of manual operations. On the other hand, our tool can do these operations automatically.
- User has to open Makefile.Agilla file and change values of some parameters. Additionally, these parameters must be set back to previous values after simulation. This process isn't necessary in our method.
- Multiple agents could be simulated with our method, while this is not possible with the other method.

- Our method can allow visualization of simulations, while the other method cannot.
- Our method can run series of simulations with autorun property of TinyViz, while the other method cannot.
- Simulation results can be recorded to files for analysis in our method. Their method can't provide recording of log files during simulations.

4.2. IMPORTANT BUG FIXES

Although we used the last version of Agilla for this project, it has many bugs. These bugs inhibit compilation, operation and simulation of Agilla. Considerable amount of time was used to fix these bugs and to acquire a stable version of Agilla as a basis for the thesis. Some of them are brought up in Agilla Forums (http://www.cse.wustl.edu/agilla/bb/) by different Agilla users and many of them have been solved by us. Reverse engineering is used for the purpose of fixing these bugs, because of lack of implementation documentation and comments on codes. The bugs and the proposed fixes to these bugs are explained in Appendix B. in detail.

4.3. ADDED INSTRUCTIONS

Agilla allows adding new instructions to its ISA. We composed new instructions for Agilla in order to reduce the code size of mobile agents. Reducing code size means faster agents.

As mentioned in Section 2.4.4.2, most important characteristic of a mobile agent is migration (moving or cloning from one node to another node). Hence, migration instructions are frequently used in mobile agents. Migration instructions are smove, wmove, sclone and wclone as mentioned in Section 3.5.1.3.

If an agent is to migrate from a source node to a destination node, the (x,y) location information of the destination node must be in the stack of the agent.

pushloc instruction is used to push the location of a node to the stack. Syntax of this instruction is *pushloc x y* where (x,y) is the location that is placed into the stack. Therefore, *pushloc* instruction must be used before migration instructions.

For example, the following instruction chain must be used for "strong move to a node at location (3,2)":

" pushloc 3 2
smove "

Each of the above instructions is one byte, so two instructions are two bytes. If there is a need to use an extra instruction, these can be combined to reduce the code size. On account of this idea, we composed four new instructions. These are smovel, wmovel, sclonel and wclonel. They are explained in Table 4.1.

Table 4.1: Added instructions to Agilla's ISA

Function	**Old Instructions**	**New Instruction**
Strong move to node at (x,y) location	*1 . pushloc x y* *2. smove*	*1. smovel (x,y)*
Weak move to node at (x,y) location	*1 . pushloc x y* *2. wmove*	*1. wmovel (x,y)*
Strong clone to node at (x,y) location	*1 . pushloc x y* *2. sclone*	*1. sclonel (x,y)*
Weak clone to node at (x,y) location	*1 . pushloc x y* *2. wclone*	*1. wclonel (x,y)*

The old instructions consume two bytes, but the new instructions consume one byte each, hence the decreased size of migration instructions decreases code size and migration delays of the agents.

CHAPTER 5

SIMULATION STUDIES

This chapter presents our proposed method for simulation of Agilla. First, benefits of simulations will be listed. Later, our developed method for simulation of Agilla will be described and working steps of Agilla Simulator will be presented under six subheadings. Afterwards, simulations which compare important instructions are presented. Then, comparison of simulations and experiments using real motes are explained.

5.1. BENEFITS OF SIMULATIONS

A WSN simulator is software which mimics the operation of the WSN. These are used to analyze and justify theoretical models and they play an important role in both industry and academia as a needed research tool. This section focuses on the major benefits presented by WSN simulators.

Major benefits of simulations:

- Simulations provide practical feedback for designing real world systems. A WSN designer can determine the effectiveness and efficiency of a design before the WSN is actually built. As a result, the WSN designer can examine the advantages of alternative designs without physically constructing the WSN.

- Cost of a simulation is much lower than building a real WSN. Furthermore, simulation tests are faster than performing real experiments because multiple tests can be handled by simulations.

- Visual simulators such as TinyViz use computer graphics and animation intelligently. Such simulators can demonstrate operation, behavior and relationship of all the simulated WSN nodes in a dynamic manner. This feature provides an expressive understanding of the WSN's character. Additionally, simulations can be used for demonstrating WSN concepts to students.

- Simulations can provide results that are not experimentally measurable. For instance, testing an application on a WSN which consist of thousands of notes is extremely difficult with real nodes. However, we can test the application on thousands of virtual nodes by the TOSSIM simulator (Zia and Zomaya, 2005). We can also easily test the application with different parameters such as a radio model. Simulations can be set to run for as many time steps and any level of detail desired as. Carley and Prietula (1994) point out; one of the main advantages of simulation models is the ability of the simulation to capture more of the real world.

- Elson et al. (2003) states that another benefit of simulation is that it provides complete visibility into the WSN being tested. Simulations provide practically unlimited space for saving sensor inputs, debugging messages, or any other information useful for comprehension of the system’s behavior.

5.2. THE SIMULATION ENVIRONMENT

To setup the simulation environment, some software must be installed. First, prerequisite software must be installed. These are: Sun’s Java Development Kit (JDK) version 1.4.x, Sun’s javax.comm package and Graphviz visualization tool.

Later, TinyOS, Agilla, TinyViz and Agilla Simulator can be installed. Steps of the setting up the simulation software are explained in Appendix C.

Network topology, routing algorithm, radio model and parameters used in simulations are important for better understanding of the simulation environment.

5.2.1. Network Topology

The simulations are performed on a 25-node network arranged in a 5x5 grid as seen in Figure 5.1.

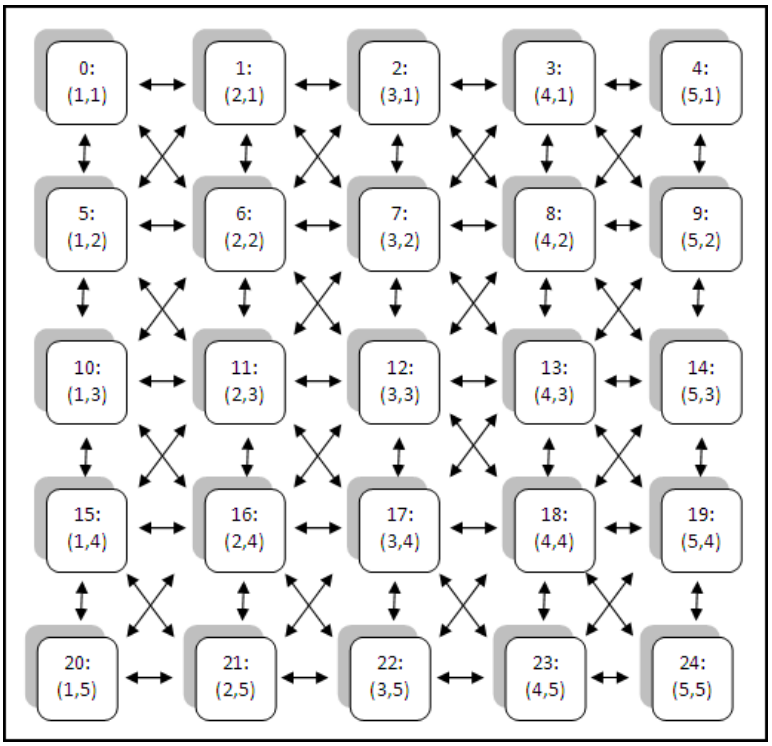

Figure 5.1: Grid topology of simulated system

In grid topology, a node can only communicate with its horizontal, vertical, and diagonal neighbors. Messages from non-neighbors are filtered-out.

5.2.2. Routing

A simple greedy-forwarding algorithm is used in simulation which is implemented by Agilla. First, each agent looks at its neighbors in this algorithm. Then, it determines the neighbor which is closer to the ultimate destination and forwards the message to this neighbor. If there is no such a neighbor, the migration and tuple space operation will fail.

5.2.3. Radio Model

The radio model plugin of TinyViz which is mentioned in Section 3.2. is used for radio simulation. This plugin sets the bit error rate between nodes considering their location and various models of radio connectivity. This model provides two realistic connectivity models: "Empirical" and "Fixed radius". Empirical model based on Mica family mote's RFM TR1000 radio is used for simulations in this study.

Nodes are placed in a directed graph in the empirical model. Each edge *(a,b)* has a value between 0 and 1 which represents the probability that a bit sent by *a* will be corrupted when b hears it. Therefore, a value of 1.0 means every bit will be corrupted while a value of 0.0 means every bit will be transmitted without error. Loss rates of this model observed empirically in an experiment performed by Ganesan et al. (2003) on a TinyOS network. Simulator can capture packet losses, data corruptions and interference by specifying error at the bit level. However, the empirical model cannot model noise; if no node transmits, every node will hear a perfectly clear channel.

5.2.4. Simulation Parameters

Simulation parameters are explained in Appendix C.

5.3. SIMULATIONS

5.3.1. Benchmarking Performance of Important Instructions

5.3.1.1. Strong Migration vs. Weak Migration

The term “migration” was mentioned in Section 3.5.1.3. In order to benchmark strong and weak migrations, we used test agents. We measured consumed time during the smove (strong move), wmove (weak move), sclone (strong clone) and wclone (weak clone) operations.

We used two different scenarios to benchmark move (smove and wmove) and clone (sclone and wclone) instructions. The difference between the scenarios is the heap operations; 10 variables are recorded to the heap in the first scenario while heap operations were not used in the second scenario. The heap is a random-access storage area that can store up to 12 variables as mentioned in Section 3.5.5.

First, we test smove and wmove instructions for each scenario (with and without heap operations). Then, sclone and wclone instructions are simulated for same scenarios.

smove vs. wmove (with heap operations)

The agent used to simulate smove instruction with heap operations is shown in Figure 5.2. In short, this agent saves 10 values to its heap and moves to a random neighbor carrying its code, program counter, stack and heap.

Figure 5.3. shows code of the wmove agent with heap operations. This agent has the same instructions as the smove agent except line #22. Again, this agent saves 10 values to its heap and moves to a random neighbor carrying only its code.

```
1:      pushc 1                 // push “1” to stack
2:      setvar 0                // record “1” to heap[0]
3:      pushc 1                 // push “1” to stack
4:      setvar 1                // record “1” to heap[1]
5:      pushc 1                 // push “1” to stack
6:      setvar 2                // record “1” to heap[2]
……
……
19:     pushc 1                 // push “1” to stack
20:     setvar 9                // record “1” to heap[9]
21:     randnbr                 // get a random neighbor
22:     smove                   // strong move to the random neighbor
23:     halt                    // agent terminates itself
```

Figure 5.2: The smove agent with heap operations

```
1:      pushc 1                 // push “1” to stack
2:      setvar 0                // record “1” to heap[0]
3:      pushc 1                 // push “1” to stack
4:      setvar 1                // record “1” to heap[1]
5:      pushc 1                 // push “1” to stack
6:      setvar 2                // record “1” to heap[2]
……
……
19:     pushc 1                 // push “1” to stack
20:     setvar 9                // record “1” to heap[9]
21:     randnbr                 // get a random neighbor
22:     wmove                   // weak move to the random neighbor
23:     halt                    // agent terminates itself
```

Figure 5.3: The wmove agent with heap operations

We simulated these agents on a virtual wireless sensor network with 25 nodes. We repeated the simulations 100 times for each agent (smove and wmove). After simulations, log files were analyzed and the average elapsed times for smove and wmove operations were computed. The results are displayed in Figure 5.4.

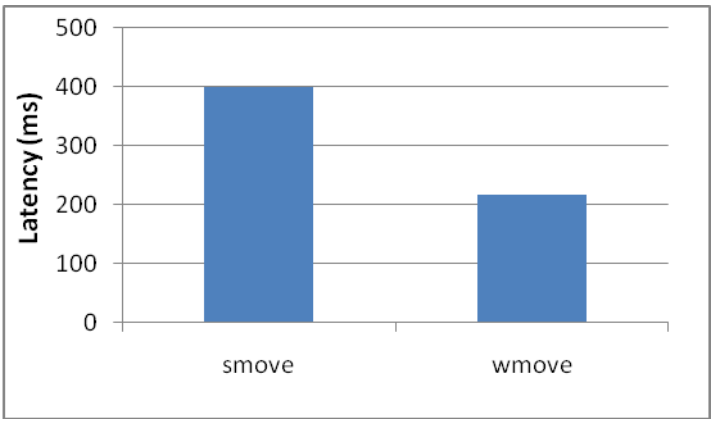

Figure 5.4: Latency of smove and wmove instructions with heap operations

According to our simulation results, average migration time of an agent from one node to another is 399 milliseconds with the smove instruction; while wmove takes 218 milliseconds, 55% of the value recorded for smove. Variances of migration times are 12.3 ms and 6.4 ms for smove and wmove respectively. The variation in the migration time is due to the noise in the network.

This result is expected, because smove instruction transfers agent's heap together with the agent's code, while wmove instruction transfers only the agent's code. Each variable in the heap is 40 bits; therefore, 10 variables use 400 bits. This extra-load increases the migration time of smove agent.

smove vs. wmove (without heap operations)

5.5. shows the smove agent and Figure 5.6. shows the wmove agent (without heap operations). In these agents, heap operations are removed to test the amount of the time taken for smove and wmove when the same amount of data is involved in both cases.

```
1:  randnbr   // get a random neighbor
2:  smove     // strong move to the random neighbor
3:  halt      // agent terminates itself
```

Figure 5.5: The smove agent without heap operations.

```
1:  randnbr   // get a random neighbor
2:  wmove     // weak move to the random neighbor
3:  halt      // agent terminates itself
```

Figure 5.6: The wmove agent without heap operations.

We also repeated the simulations for 100 times for each agent (smove or wmove). We then computed the average latency for smove and wmove operations which are shown in Figure 5.7.

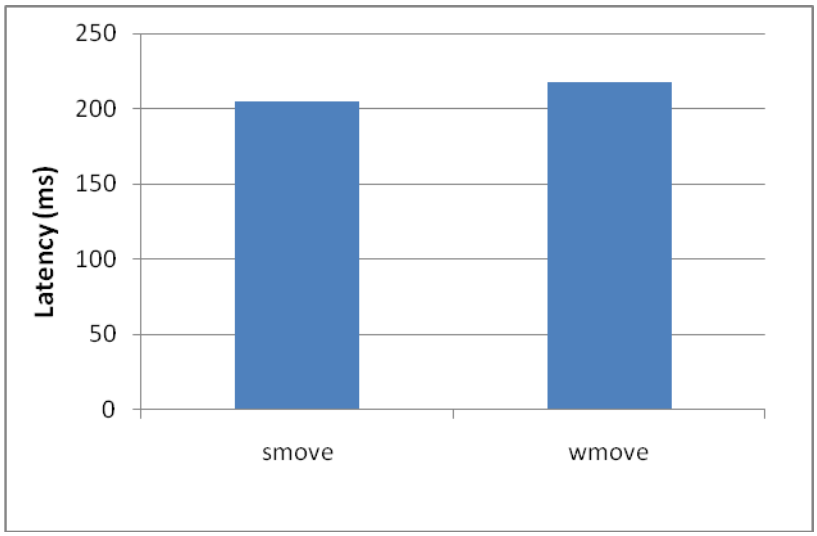

Figure 5.7: Latency of smove and wmove instructions without heap operations

There is only 6% difference between latencies of each move instruction (205 milliseconds for smove and 218 milliseconds for wmove). Variances of smove and wmove simulations are 6 and 6.4 ms respectively. As a result of these simulations, we can say that same amount of transferred data means same amount of consumed time for smove and wmove.

The other consequence of the above simulations is the migration speed of the agents. An agent can migrate once every 300 ms on average. Average radio range of Mica motes are 30 m. as mentioned in Section 2.2.1. Therefore, an agent can migrate across a network at 360 km/h. This speed is sufficient for many applications such as intruder tracking, fire tracking, vehicle tracking etc.

sclone vs. wclone (with heap operations)

Codes of the agents used to benchmark sclone and wclone instructions are similar to the agents in Figure 5.5. and Figure 5.6.; only line #22 (smove/wmove) of the agents is replaced by sclone/wclone. We repeated the simulations for 100 times on a simulated wireless sensor network with 25 nodes as in the previous simulations. Figure 5.8. displays the results of the simulations for sclone and wclone agents with heap operations.

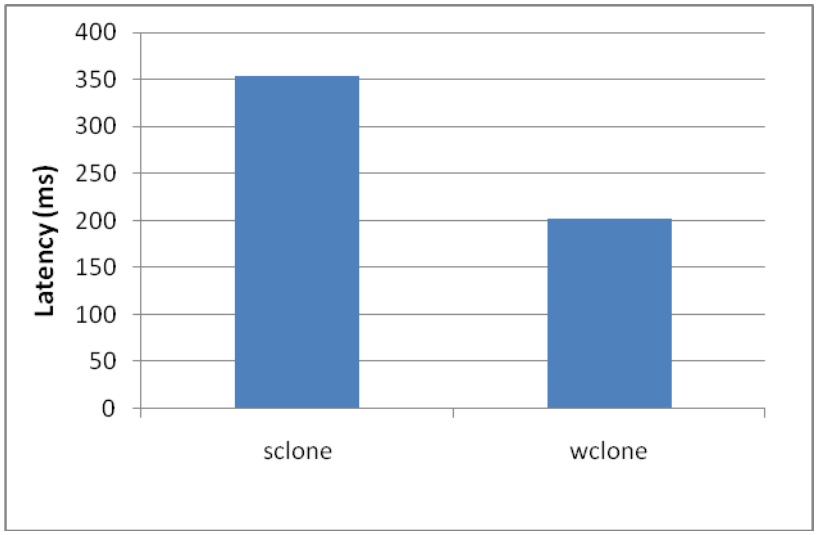

Figure 5.8: Latency of sclone and wclone instructions with heap operations

According to the results, average migration time of an agent from one node to another is 352 milliseconds with sclone instruction; while wclone takes 202 milliseconds. This result is expected, as transferring agent's code together with the agent's heap increases latency.

5.3.1.2. Spreading Benchmark

Another test agent is composed to measure the spreading performance of the agents. We created a scenario according to which an agent is injected to a node of the WSN to sense the temperature of the node and send the recorded value to the base station, and then it enters a loop. In the loop, the agent clones itself to its neighbors for spreading into the network. Neighbor nodes are chosen randomly by *randnbr* instruction. This instruction looks at the neighbor list of the agent and chooses a random location from the list.

```
1:        pushc TEMP  // indicates temperature sensor will be read
2:        sense         // get sensor reading
3:        loc           // push location of the node to stack
4:        pushc 2       // indicates top two value of stack will be sent
5:        pushloc 1 1   // push location of base station to stack
6:        rout          // send sensor reading to base station
7:        pushc 1       // push 1 to stack to indicate red led
8:        putled        // blink red led
9:      pushc 3       // push 3 to stack
10:  LOOP        dec   // decrement value in the stack
11:       copy          // copy top value of stack
12:       pushc 0       // push 0 to stack
13:       ceq           // is top two value of stack are equal?
14:       rjumpc DIE    // if the loop has done, jump to DIE
15:     randnbr         // choose a random neighbor
16:       rjumpc CLONE // if there is a neighbor, jump to CLONE
17:       rjump LOOP
18:  CLONE       wclone // if there is a neighbor, weak clone
19:       rjump LOOP    // jump to LOOP
20:  DIE          halt      // agent terminate itself
```

Figure 5.9: Spreading Agent

Table 5.1: Line by Line Explanation of the Agent in Figure 5.9.

Line	Explanation
1	Push “TEMP” on to the stack to indicate temperature sensor.
2	Get specified sensor reading on top of the stack and push it on to the stack.
3	Push location of the current node to the stack.
4	Push “2” to the stack to indicate that the top two values will be sent to the node.
5	Push location of the base station (1,1) to the stack.
6	Send temperature and location of the current node to the base station.
7	Push “1” to indicate red led (red: 1, green: 2, yellow: 4).
8	Read top value of the stack and blink specified led to indicate that the current node sent temperature to the base station.
9	Push “3” to the stack to indicate remaining number of loops.
10	Here, the word “LOOP” is a label. A label is an identifier like the target of rjumpc and rjump instructions. dec instruction decrements the top value of the stack to decrease remaining number of loops.
11	Copy top of the stack.
12	Push “0” to the stack.
13	ceq instruction compares the top two values of the stack. If values are equal set condition code to 1.
14	If conditional code is 1, it means the loop is done. Jump to label “DIE” to terminate agent. If conditional code is 0, resume from line #15.
15	Choose a random neighbor. If there is a neighbor, push its location to the stack and set condition code to 1. If not, set condition code to 0.
16	If condition code is 1, jump to label “CLONE”. If not, resume from line #17.
17	Jump to label “LOOP” whatever the condition code is.
18	Here, the word “CLONE” is a label. wclone instruction weak clones the agent to the location in the top of the stack.
19	Jump to label “LOOP”.
20	Here, the word “DIE” is a label. halt instruction terminates the agent.

The goal of the agent is to visit all the nodes of the WSN and send temperature readings to the base station. Clone operations were preferred to provide faster spreading. Since *wclone* instruction is more efficient than *sclone* as mentioned before, it was chosen for migration. Code of the agent can be seen in Figure 5.9. and detailed execution of the agent can be seen in Table 5.1.

This agent has been tested on simulated wireless sensor networks which included various number of nodes. The agent was tested 20 times on each WSN. Average values of the tests are taken as the final result. Simulation results are shown in Figure 5.10.

The graph shows that the agent can spread into a 5-node WSN in 0.4 seconds while it can spread into a 20-node WSN in 1.9 seconds. As expected, larger WSN increases spreading time. On the other hand, latency per node is decreased as the network size increases. For example, 125 ms is consumed per node for a 5-node WSN, while 95 ms. is consumed per node for a 20-node WSN.

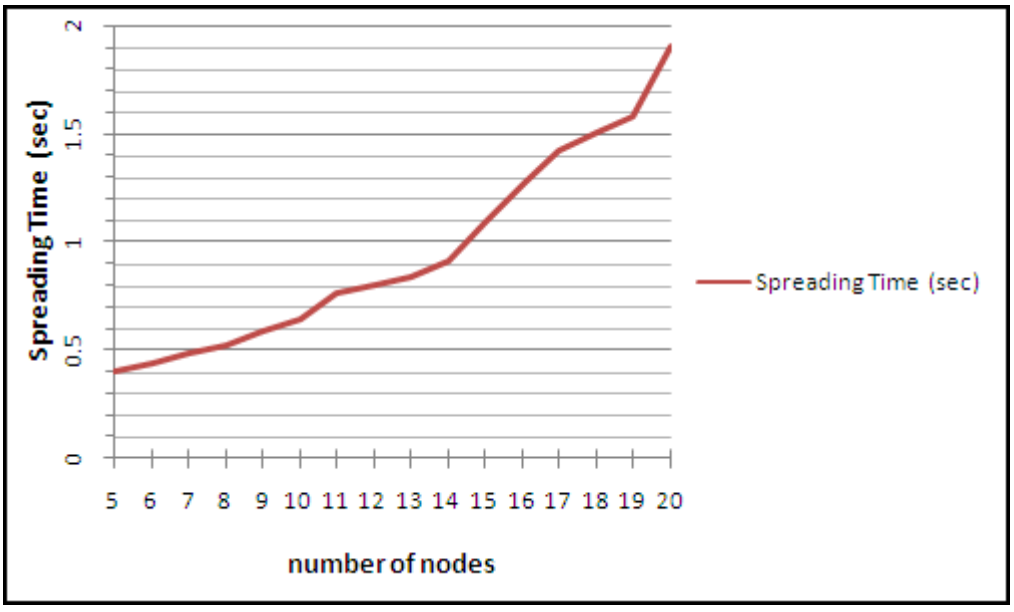

Figure 5.10: The Agent's Spreading Time into WSN

As mentioned before, the agent clones itself to its neighbors randomly. Accordingly, the agent is cloned to some nodes more than once while other nodes aren't visited by the agent. Log files are analyzed and reliability of this agent is computed according to the equation.

$$Reliability = \frac{Total\ number\ of\ nodes - Number\ of\ nonvisited\ nodes}{Total\ number\ of\ nodes}$$

Figure 5.11: Reliability equation of the agent

Result of the reliability analysis is shown in Figure 5.12. Reliability of the agent changes between 82% and 91% inconsistently because of the random nature of the migration operation.

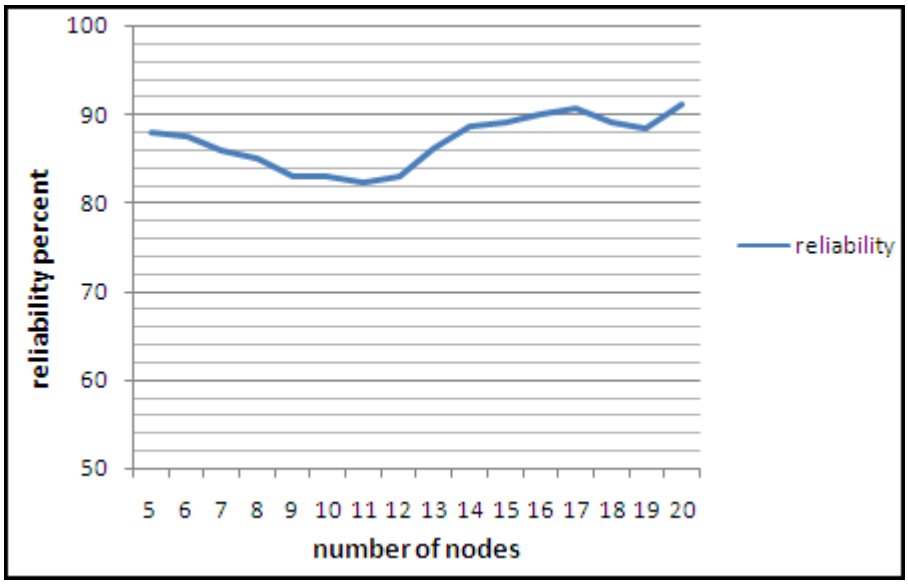

Figure 5.12: Reliability of the agent in Figure 5.9.

We improved the agent to avoid duplicate migrations and some codes are added to the agent. First, some codes are added to beginning of the agent to indicate that this node has already been visited. The agent adds a specified value to the tuple space of the current node. Second, some codes are added before the clone operation to control if the remote node has already been visited or not using the *rrdp* instruction. *Rrdp* instruction searches the remote node's tuple space for a

matching tuple with the specified value. If there is a matching value, this means the remote node has already been visited. In this case, the agent does not clone itself to the remote node and it chooses a new neighbor. Therefore, modified version of the agent checks the neighbor node before cloning itself. Thus, a node is visited only once by an agent.

Modified agent was simulated under the same conditions of the original agent. Average spreading speed of the agent is shown in Figure 5.13.

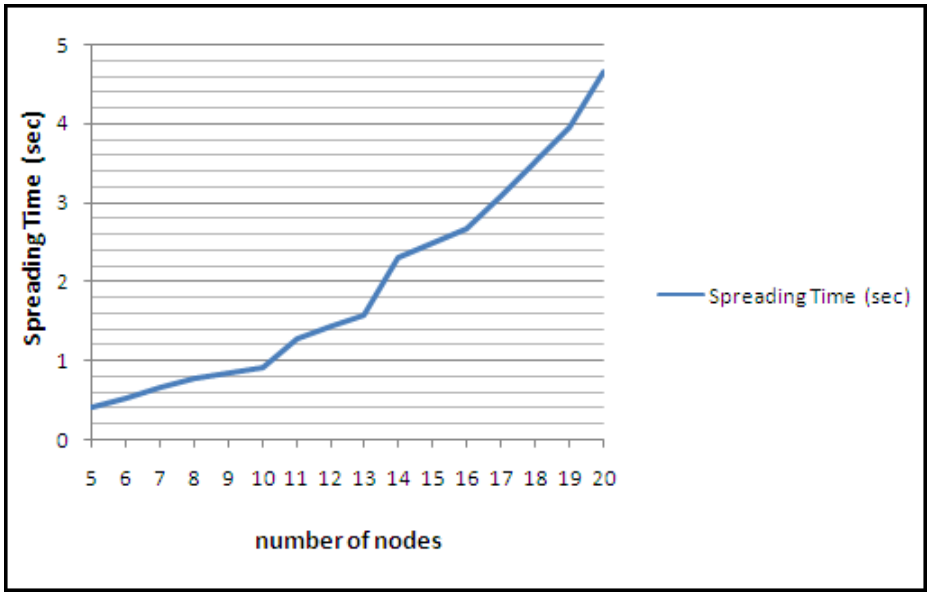

Figure 5.13: The Modified Agent's Spreading Time into WSN

Figure 5.14 is included to compare the two results. It shows that the latencies are increased for the modified agent. There are two possible reasons: First, appended instructions causes overhead in the code of the modified agent. As mentioned before, more code size means more latency. Second, the consumed time by *rrdp* instruction increases latency. Simulations and real life experiments show that latency of *rrdp* instruction is approximately 60 ms.

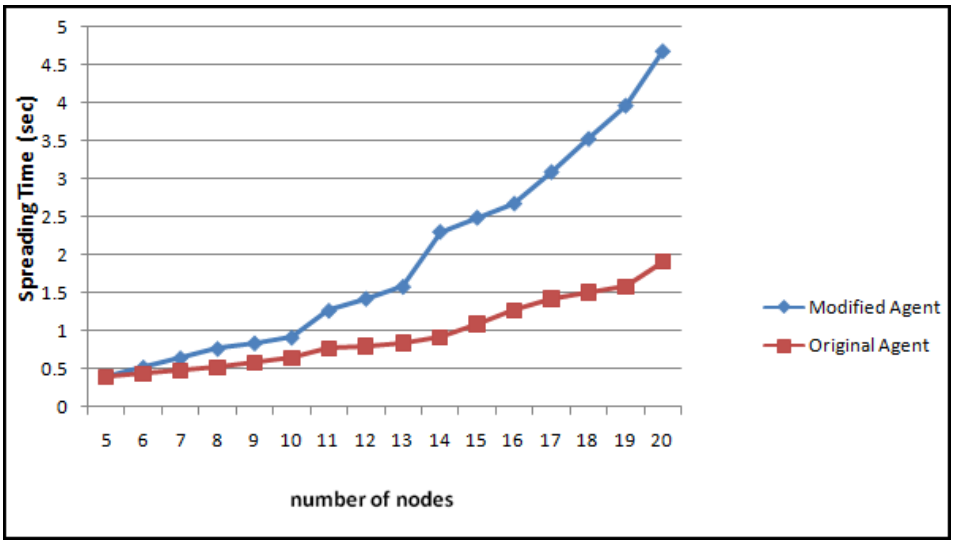

Figure 5.14: Comparison Chart of Spreading Times

Reliability of the modified agent is also computed. Comparison of reliabilities of the original and the modified agent can be seen in Figure 5.15.

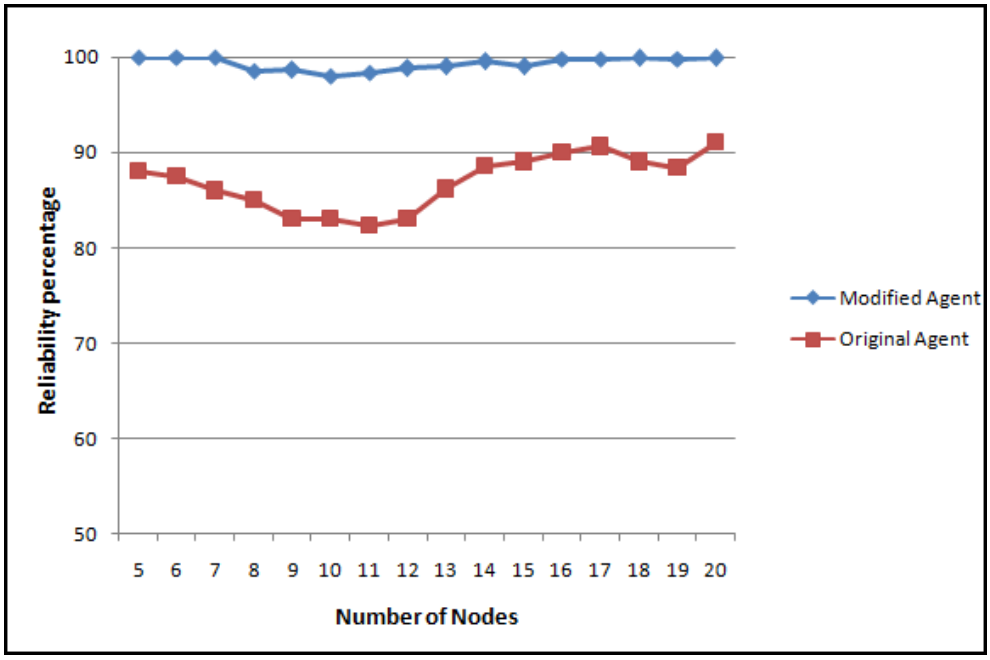

Figure 5.15: Reliability Comparison of Agents

Usage of *rrdp* operations reduces network performance, but it increases reliability as seen in Figure 5.15.

5.3.2. Comparison of Simulations with Real Life WSN Experiments

Latency and reliability of individual Agilla instructions are evaluated by Agilla developers in a technical report (Fok et al., 2006). In this study, same instructions are benchmarked on our simulation environment. Empirical results which were obtained during real experiments and simulation results are compared. Comparison of the outcomes of the simulated system and real system are presented in this section.

Fok et al. (2006) used a 25-node network formed into a 5x5 grid topology in real life experiments as shown in Figure 5.13. In these experiments, nodes are identified by (x,y) coordinates based on their grid positions. All messages except those from the neighbors are filtered in the grid topology for simulating multi-hop routing. Additionally, a simple best-effort greedy forwarding algorithm is used to provide geographic routing.

We compared the results of simulations and the results of real experiments to evaluate the usability of mobile agent simulations. Same number of nodes, same network topology and same agents are used for consistency.

First, agent migration operations and then, remote tuple space operations are examined.

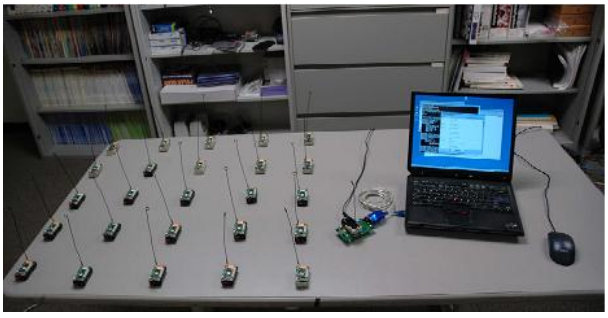

Figure 5.16: 5x5 mote test bed which is used in real experiments (Fok et al., 2006)

Smove

In this simulation, the smove agent, shown in Figure 5.17., moves from node (1,1) to a remote node, then turns back. Therefore, the agent migrates two times. Remote node could be 1 to 5 hops away. This experiment was repeated 100 times for each number of hops. In other words, 500 simulations were made for this agent. Average latency of successful executions and number of failures are calculated after analyzing the log files (latencies are halved to consider double migration).

```
1:    pushloc 5 1      // push location (5,1) to stack
2:    smove            // strong move to node at (5,1)
3:    pushloc 0 0      // push location (0,0) to stack
4:    smove            // strong move to node at (0,0)
5:    halt             // agent terminates itself
```

Figure 5.17: Code of smove agent

Comparison graph of average latency of this simulation and average latency of real experiment is shown in Figure 5.18.

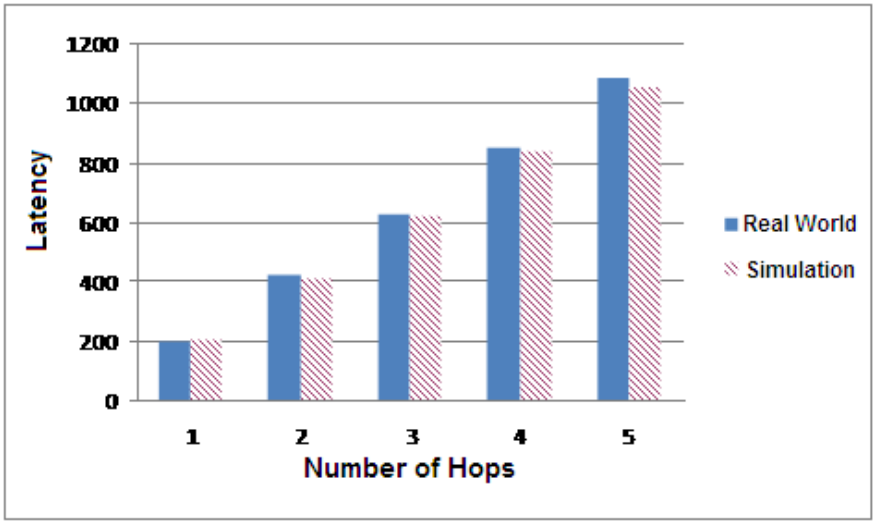

Figure 5.18: Comparison of simulations and real experiments results of latency of smove instruction.

According to the figure, differences between values of the simulation environment and real experiments differ up to 3%. Therefore, the results of the simulations and real experiments are almost equal. This outcome indicates that the simulation environment and real world environment produces very similar results for latency of smove agent. However, further simulations must be made to validate the accuracy of the simulation environment.

Detailed statistical information such as minimum, maximum and average values and variances of simulations are also shown in Table 5.2.

Table 5.2: Min, max and average values and variances for smove agent

Number of Hops	**1**	**2**	**3**	**4**	**5**
Minimum (ms)	196	392	588	795	1003
Maximum (ms)	214	430	649	878	1099
Average (ms)	205	411	619	838	1054
Variance (ms)	6.1	11.6	17.7	22.6	30.3

Reliability of smove instruction is also calculated. Reliability of an instruction is computed using the following equation:

$$Reliability = \frac{Total\ number\ of\ executions - Number\ of\ failed\ executions}{Total\ number\ of\ executions}$$

Figure 5.19: Reliability formula of an instruction

Because of unreliable wireless links of a WSN, a middleware should use some techniques to increase network reliability. Fok et al. (2006) states that Agilla uses two techniques. A simple protocol, which involves acknowledgements, timers and retransmits, is used in the first technique. In the second technique, all of the message packets are transmitted and acknowledged hop by hop. This approach prevents re-sending of a message through multiple hops when it is lost.

Comparison of reliability of smove instruction is shown in Figure 5.20. According to the figure, difference between reliability values of the simulation environment and real environment changes up to 3%. This means the simulation system can successfully mimic packet losses of the real WSN system, because reliability is mainly related to packet losses. Moreover, Figure 5.20 shows that increasing the distance decreases reliability.

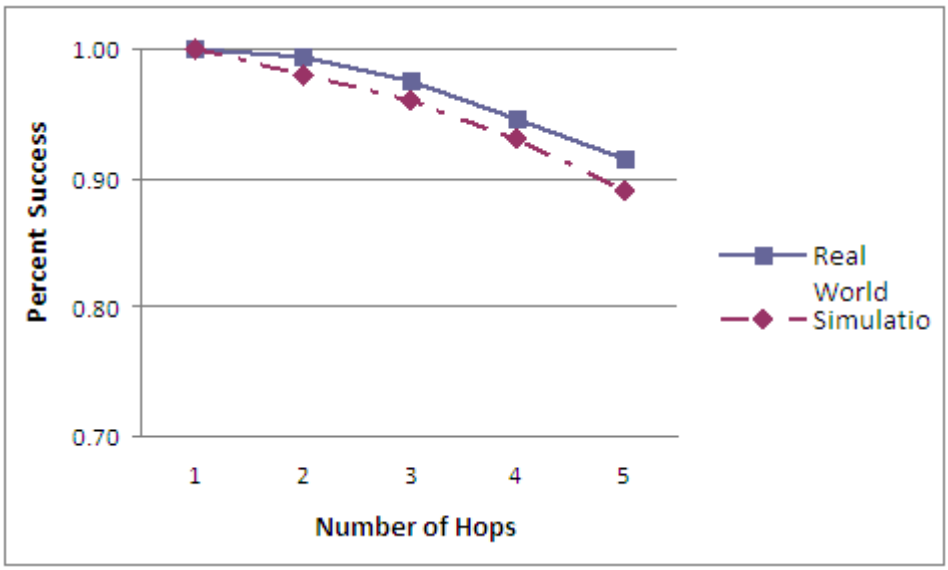

Figure 5.20: Comparison of the results of simulations and real experiments of reliability of smove instruction.

Rout

Rout (remote out) agent, shown in Figure 5.21., inserts a tuple in a remote node's tuple space. Simulations are performed to test this operation and tests are repeated 100 times for 1 to 5 hops. Comparisons of simulation results with real world experiments are shown in Figure 5.22 and Figure 5.23. Detailed statistical information is also shown in Table 5.3.

```
1:  pushloc 5 1     // push location (5,1) to stack
2:  smove           // strong move to node at (5,1)
3:  pushloc 0 0     // push location (0,0) to stack
4:  smove           // strong move to node at (0,0)
5:  halt            // agent terminates itself
```

Figure 5.21: Code of rout agent

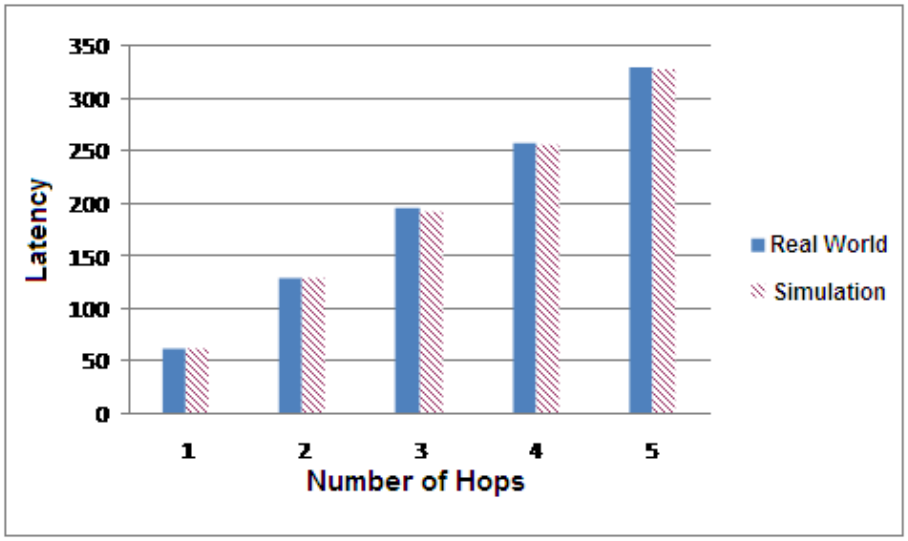

Figure 5.22: Comparison of the results of simulations and real experiments of latency of rout instruction

Figure 5.22. shows that simulated system produces same results with real system for rout agent. Difference between of the results of the simulated system and the real system is under 1%.

Table 5.3: Min, max and average values and variances for rout agent

Number of Hops	**1**	**2**	**3**	**4**	**5**
Minimum (ms)	58	121	184	241	308
Maximum (ms)	64	133	201	268	342
Average (ms)	61	127	192	255	326
Variance (ms)	2.1	3.5	6.1	8.2	9.3

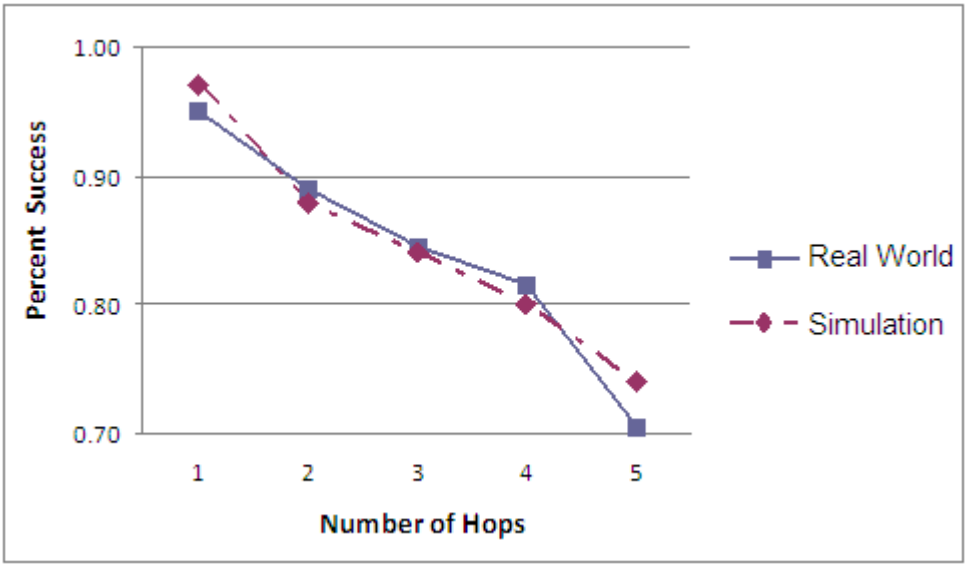

Figure 5.23: Comparison of simulations and real experiments results of reliability of rout instruction

Figure 5.23 shows that reliability values of the simulation environment and real environment are similar. Apart from this, it is seen that increasing distance increases packet losses and decreases reliability.

Latency and reliability results of smove and rout agents point out that there is a tradeoff between latency and reliability. Rout agent has a lower latency than smove agent. On the other hand, smove agent is more reliable than rout agent.

Remote Operations

Agilla middleware enables remote coordination of agents by remote operations. Most important ones are rout, rinp, rrdp (remote probing rd) and migration instructions (smove, wmove, sclone, wclone).

Rout instruction and migration instructions have previously been mentioned. Rinp (remote probing in) and rrdp (remote probing rd) search remote node's tuple space for a matched template. If a matched template is found, rinp removes the matched tuple from remote node's tuple space, while rrdp does not.

Comparisons of simulation results of remote operations with those obtained in real life experiments are shown in Figure 5.24. According to the figure, simulations produce similar results with real experiments for both remote tuple space operations (rinp, rout and rrdp) and migration operations (smove, wmove, sclone, wclone). However, results of simulations are consistently lower than results of real experiments (1% to 5%), because empirical radio model used in simulations cannot model noise as mentioned Section 5.2.

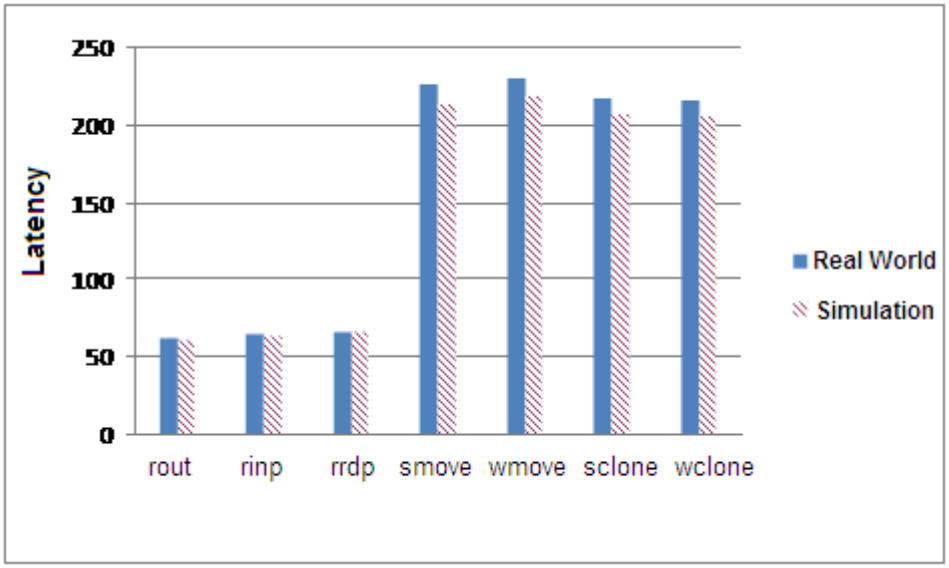

Figure 5.24: Comparison of results of the simulations and real experiments of latency of remote operations.

Table 5.4: Min, max and average values and variances for remote operations

Instruction	rout	rinp	rrdp	smove	wmove	sclone	wclone
Minimum (ms)	58	60	61	204	208	197	196
Maximum (ms)	64	67	68	223	227	214	214
Average (ms)	61	63	65	213	218	206	205
Variance (ms)	2.2	2.1	2.3	5.8	5.3	5.1	5.3

Detailed statistical information about remote operations is shown in Table 5.4.

CHAPTER 6

CONCLUSIONS

In this chapter, summary of the work done and the major contributions of the study will be discussed. Finally, the chapter concludes with recommendations for further research.

6.1. SUMMARY OF THE WORK DONE

In this study, state-of-the-art middleware for wireless sensor networks were evaluated and classified in four categories. The role of middleware, as well as the advantages and disadvantages of each approach were presented. Moreover, major middleware which belong to different approaches were analyzed and their strong and weak points were determined. It was determined that, the agent based approach and Agilla middleware offers various advantages to the user.

The existing Agilla application was analyzed. It had problematic bugs some of which were blocking the compilation of Agilla. These bugs were fixed, and Agilla developers were informed about them, as well as the corresponding fixes. As a result of this study, a functioning version of Agilla can now be supplied.

In addition to bug fixes, useful instructions were added to the existing Agilla applications. These new instructions combine some existing instructions in order to reduce the size of the agent's code. Reduced code size implies faster execution of an agent.

A first effort is made for simulation of mobile agents running on Agilla using a software application. A Java application, which is called Agilla Simulator, was developed in order to transfer Agilla to the simulation environment. A user can write and simulate an agent and save log files for offline analysis via the Agilla Simulator.

Simulation experiments were made using the Agilla Simulator. First goal in conducting the simulations was to measure performance of important Agilla instructions.

The other and most important objective of the simulations was to compare the performance of a simulated system with a real system. This study is significant for the purpose of validating simulation tools. Since we do not have real motes, we considered micro-benchmarks experimented on real motes by Fok et al. (2006). Simulation results show that simulated system produces almost the same results as the real system (Özarslan & Erten, 2008).

6.2. FURTHER RESEARCH

Further research activities can be classified into different areas. First of all, the functionality of Agilla middleware can be enhanced. Due to limited resources of sensor nodes, existing security algorithms are not suitable for WSNs. Agilla does not support security and QoS aspects, like most of the other middleware applications. In order to improve Agilla, efficient security models can be developed. Moreover, some mechanisms can be built to provide QoS parameters such as bandwidth allocation and ensuring reliable service.

Simulations can be diversified. Different network topologies can be simulated. For example, simulations can be repeated on different number of nodes with different distances among them. Furthermore, different simulation tools such as OMNET++ and NS-2 can be used for simulations, and performance of simulators can be evaluated.

REFERENCES

Akyildiz, I., Su, W., Sankarasubramaniam, Y., & Cayirci, E. (2002). Wireless sensor networks: a survey. *Computer Networks, 38*(4), 393–422. doi:10.1016/S1389-1286(01)00302-4

Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., et al. (2004). A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks, 46*(5), 605-634. doi:10.1016/j.comnet.2004.06.007

Asada, G., Dong, M., Lin, T. S., Newberg, F., Pottie, G., Kaiser, W. J., et al. (1998). Wireless integrated network sensors: Low power systems on a chip. *Proceedings of the 24th European Solid-State Circuits Conference*, 9-16.

Barr, R., Bicket, J. C., Dantas, D. S., Du, B., Kim, T. W. D., Zhou, B., et al. (2002). On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Operating Systems Review, 36*(2), 1-5. doi: 10.1145/509526.509528

Boulis, A., Han, C. C., & Srivastava, M. B. (2003). Design and implementation of a framework for efficient and programmable sensor networks. *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, 187-203. doi: 10.1145/1066116.1066121

Cabri, G., Leonardi, L., & Zambonelli, F. (2000). MARS: a programmable coordination architecture for mobile agents. *Internet Computing, IEEE, 4*(4), 26-35. doi: 10.1109/4236.865084

Carley, K. M., & Prietula, M. J. (1994). *Computational organization theory*. Mahwah, NJ, USA: Lawrence Erlbaum Associates Inc.

Chess, D., Grosof, D., Harrison, C., & Levine, D. (1995). Itinerant agents for mobile computing. *IEEE Personal Communications, 2*(5), 34-49.

Chess, D., Harrison, C., & Kershenbaum, A. (1997). Mobile agents: Are they a good idea? *Lecture Notes In Computer Science, 1222*, 25-45.

Cugola, G., & Picco, G. P. (2001). *Peerware: Core middleware support for peer-to-peer and mobile systems*. Milano: Dipartimento di Electronica e Informazione, Politecnico di Milano.

Elson, J., Bien, S., Busek, N., Bychkovskiy, V., Cerpa, A., Ganesan, D., et al. (2003). *EmStar: An environment for developing wireless embedded systems software*. Los Angeles, CA, USA: Center for Embedded Networked Sensing (CENS).

Elson, J., Gansner, E., Koutsofios, L., North, S. C., & Woodhull, G. (2002). Graphviz-open source graph drawing tools. *Lecture Notes in Computer Science, 2265*, 594-597.

Fok, C. L., Roman, G., & Lu, C. (2006). *Agilla: a mobile agent middleware for sensor networks*. St. Louis, MI, USA: Washington University in St. Louis.

Fok, C. L., Roman, G. C., & Lu, C. (2005). Mobile agent middleware for sensor networks: an application case study. *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, 382-387.

Ganesan, D., Krishnamachari, B., Woo, A., Culler, D., Estrin, D., & Wicker, S. (2003). *Complex behavior at scale: An experimental study of low-power wireless sensor networks*. Los Angeles, CA, USA: UCLA Computer Science.

Gao, Q., Blow, K. J., Holding, D. J., Marshall, I. W., & and Peng, X. H. (2006). Radio range adjustment for energy efficient wireless sensor networks. *Ad Hoc Networks, 4*(1), 75-82. doi:10.1016/j.adhoc.2004.04.007

Gay, D., Levis, P., Behren, R. v., Welsh, M., Brewer, E., & Culler, D. (2003). The nesC language: A holistic approach to networked embedded systems. *ACM SIGPLAN Notices, 38*(5), 1-11. doi: 10.1145/781131.781133

Gelernter, D. (1985). Generative communication in Linda. *ACM Transactions on Programming Languages and Systems, 7*(1), 80-112.

Gutierrez, J. A., Naeve, M., Callaway, E., Bourgeois, M., Mitter, V., & Heile, B. (2001). IEEE 802.15. 4: a developing standard for low-power low-costwireless personal area networks. *Network, IEEE, 15*(5), 12-19. doi: 10.1109/65.953229

Hackmann, G., Fok, C. L., Roman, G. C., Lu, C., Zuver, C., English, K., et al. (2005). Demo abstract: Agile cargo tracking using mobile agents. *Proceedings of the 3rd Annual Conference on Embedded Networked Sensor Systems*, 303. doi: 10.1145/1098918.1098968

Hadim, S., & Mohamed, N. (2006a). Middleware for wireless sensor networks: a survey. *Proceedings of First International Conference on Communication System Software and Middleware*, 1-7.

Hadim, S., & Mohamed, N. (2006b). Middleware: middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online, 7*(3). doi:10.1109/MDSO.2006.19

Han, Q., & Venkatasubramanian, N. (2001). AutoSeC: an integrated middleware framework for dynamic service brokering. *IEEE Distributed Systems Online, 2*(7).

Herbert, J., O'Donoghue, J., Ling, G., Fei, K., & Fok, C. L. (2006). Mobile agent architecture integration for a wireless sensor medical application. *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology*, 235-238. doi: 10.1109/WI-IATW.2006.90

Hill, B. M., Bacon, J., Mako, B., Burger, C., Jesse, J., & Krstic, I. (2006). *The official Ubuntu book*. Upper Saddle River, NJ, USA: Prentice Hall.

Hill, J. L., & Culler, D. E. (2002). Mica: a wireless platform for deeply embedded networks. *IEEE Micro, 22*(6), 12-24. doi: 10.1109/MM.2002.1134340

Johansen, D., & Schneider, F. B. (1995). *An introduction to the TACOMA distributed system version 1.0*. Tromsø: Universitetet i Tromsø.

Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., & Iftode, L. (2004). Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal, 47*(4), 475-494. doi: 10.1093/comjnl/47.4.475

Lange, D. B., & Oshima, M. (1999). Seven good reasons for mobile agents. *Communications of the ACM, 42*(3), 88-89.

Levis, P., & Culler, D. (2002). Mate: a tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review, 36*(5), 85-95. doi: 10.1145/605397.605407

Levis, P., & Lee, N. (2003). *TOSSIM: a simulator for TinyOS networks*. Berkeley, CA, USA: Computer Science Division, University of California Berkeley.

Levis, P., Lee, N., Welsh, M., & Culler, D. (2003). TOSSIM: accurate and scalable simulation of entire TinyOS applications. *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, 126-137. doi: 10.1145/958491.958506

Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., et al. (2005). TinyOS: an operating system for wireless sensor networks. In J. R. W. Weber, and E. Aarts (Ed.), *Ambient Intelligence*. New York, NY, SA: Springer-Verlag.

Luk, M., Mezzour, G., Perrig, A., & Gligor, V. (2007). MiniSec: a secure sensor network communication architecture. *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, 479-488. doi: 10.1145/1236360.1236421

Madden, S. R., Franklin, M. J., Hellerstein, J. M., & Hong, W. (2005). TinyDB: An acquisitional query processing system for sensor networks. ACM Transactions on Database Systems (TODS), 30(1), 122-173. doi: 10.1145/1061318.1061322

Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., & Anderson, J. (2002). Wireless sensor networks for habitat monitoring. *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, 88-97. doi: 10.1145/570738.570751

Malan, D., Fulford-Jones, T., Welsh, M., & Moulton, S. (2004). CodeBlue: An ad hoc sensor network infrastructure for emergency medical care. *Proceedings of the MobiSys 2004 Workshop on Applications of Mobile Embedded Systems (WAMES 2004)*, 12-24.

Molla, M. M., & Ahamed, S. I. (2006). A survey of middleware for sensor network and challenges. *Proceedings of the 2006 International Conference Workshops on Parallel Processing*, 223-228. doi: 10.1109/ICPPW.2006.18

Murphy, A., & Heinzelman, W. (2002). *Milan: Middleware linking applications and networks*. Rochester, NY, USA: University of Rochester.

Noer, G. J. (1998). Cygwin: a free Win32 porting layer for UNIX applications. *Proceedings of the 2nd USENIX Windows NT Symposium*, 31-38.

Özarslan, S., & Erten, Y. M. (In press). *Simulation of Agilla middleware on TOSSIM*. Paper will be presented at the Proceedings of the First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems. SIMUTOOLS 2008.

Polastre, J., Szewczyk, R., & Culler, D. (2005). Telos: enabling ultra-low power wireless research. *Fourth International Symposium on Information Processing in Sensor Networks, 2005. IPSN 2005.*, 364-369. doi: 10.1145/1127777.1127833

Roman, R., Zhou, J., & Lopez, J. (2005). On the security of wireless sensor networks. *Proceedings of International Conference on Computational Science and Its Applications*, 9-12.

Romer, K. (2004). Programming paradigms and middleware for sensor networks. *Proceedings of GI/ITG Workshop on Sensor Networks*, 49-54.

Romer, K., Kasten, O., & Mattern, F. (2002). Middleware challenges for wireless sensor networks. *Proceedings of ACM SIGMOBILE Mobile Computing and Communications Review, 6*(4), 59-61. doi: 10.1145/643550.643556

Shen, C. C., Srisathapornphat, C., & Jaikaeo, C. (2001). Sensor information networking architecture and applications. *IEEE Personal Communications, 8*(4), 52-59. doi: 10.1109/98.944004

Shen, X., Wang, Z., & Sun, Y. (2004). Wireless sensor networks for industrial applications. *Proceedings of Fifth World Congress on Intelligent Control and Automation., 4*, 3636-3640. doi: 10.1109/WCICA.2004.1343273

Shin, K. Y., Song, J., Kim, J. W., Yu, M., & Mah, P. S. (2007). REAR: Reliable energy aware routing protocol for wireless sensor networks. *Proceedings*

of the 9th International Conference on Advanced Communication Technology, 1, 525-530.

Souto, E., Guimaries, G., Vasconcelos, G., Vieira, M., Rosa, N., Ferraz, C., et al. (2006). Mires: a publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing, 10*(1), 37-44. doi: 10.1007/s00779-005-0038-3

Wenjie, C., Lifeng, C., Zhanglong, C., & Shiliang, T. (2005). A realtime dynamic traffic control system based on wireless sensor network. *Proceedings of International Conference Workshops on Parallel Processing, 2005*, 258-264. doi: 10.1109/ICPPW.2005.16

Yao, Y., & Gehrke, J. (2002). The Cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record, 31*(3), 9-18. doi: 10.1145/601858.601861

Yu, Y., Krishnamachari, B., & Prasanna, V. K. (2004). Issues in designing middleware for wireless sensor networks. *IEEE Network, 18*(1), 15-21. doi: 10.1109/MNET.2004.1265829

Zia, T., & Zomaya, A. (2005). An analysis of programming and simulations in sensor networks. *Proceedings of the 1st International Workshop on Sensor Networks and Applications*, 15-19.

APPENDICES

APPENDIX A. SAMPLE TINYVIZ AUTORUN FILES USED IN SIMULATIONS

The autorun file in Figure A.1. runs one simulation on 5 nodes which is logged to a file. TinyViz run the simulation for 40 simulation seconds. After finishing the simulation, TinyViz automatically terminates itself.

```
# Set the layout (grid, random, grid+random)

layout grid

# This plugin takes debug messages

plugin DebugMsgPlugin

# Total number of simulated seconds to run

numsec 40

# Name of the executable file

Executable /opt/tinyos-
1.x/contrib/wustl/apps/Agilla/build/pc/main.exe

# Number of nodes

nummotes 5

# File to log all debug messages to

logfile sim1_25node
```

Figure A.1. A sample autorun file which is used in simulations.

The autorun file in Figure A.2. runs 16 simulations on 5 to 20 nodes. Each simulation is logged to a different file. TinyViz runs each simulation for 100 simulation seconds. After finishing all of the simulation, TinyViz automatically terminates

```
# Set the layout (grid, random, grid+random)

layout grid

# This plugin takes debug messages

plugin DebugMsgPlugin

# Total number of simulated seconds to run

numsec 100

# Name of the executable file

Executable /opt/tinyos-
1.x/contrib/wustl/apps/Agilla/build/pc/main.exe

# Number of nodes

nummotes 5

# File to log all debug messages to

logfile sim4_5node

# The blank line above indicates starting another simulation


# This time run with a different number of motes

Nummotes 6

logfile sim4_6node


Nummotes 7

logfile sim4_6node

……

Nummotes 20

logfile sim4_20node
```

Figure A.2. The autorun file which is used in spreading simulations.

APPENDIX B. IMPORTANT BUG FIXES

Incorrect File Paths

Initially, file paths in Agilla is adjusted for Cygwin/Windows environment. In later versions, file paths are defined for both Cygwin and Linux. However, some file paths are still adjusted for only Cygwin. This problem causes important errors.

For example, if you try to run AgentInjector on a Linux operating system like Ubuntu, you may encounter the following error:

java.io.FileNotFoundException: C:\Program Files\UCB\cygwin\opt\tinyos-1.x\contrib\wustl\apps\AgillaAgents\Tests/GetAgents.ma

You need to find agilla.properties file and replace *"initDir=C:\\Program Files\\UCB\\cygwin\\opt\\tinyos-1.x\\contrib\\wustl\\apps\\AgillaAgents\\Tests"* line with *" initDir=../../apps/AgillaAgents "* to solve this problem.

Misspelling of Some Words on Codes

Other frequently encountered errors are about misspelling of some words in the codes. For instance, the following error is encountered during the compilation of Agilla: *"types/LocationDirectory.h:5:21: TOSTime.h: No such file or directory"*

Correct name of "*TOSTime.h*" file is "*TosTime.h*". In order to solve this problem, the line *#include "TOSTime.h"* of "LocationDirectory.h" file should be replaced with *#include "TosTime.h"*

Unused Variable Errors

Another frequently encountered error type is the unused variable error. This type of error does not block compiling of Agilla. Some created variables were forgotten to be removed even though they are now useless. As an example, the

compiler error in Figure B.1. cannot block compiling of Agilla, but it is inconvenient. User should delete the following line from LocationReporterM.nc file to avoid the error message:

"struct AgillaLocMsg *sMsg = (struct AgillaLocMsg *)msg->data;"

```
components/LocationReporter/LocationReporterM.nc: In function
`LocationReporterM$sendMsg':
components/LocationReporter/LocationReporterM.nc:158: warning: unused
variable `sMsg' "
```

Figure B.1. Compile Error of "sMsg" variable bug

CC2420Control Bug

As stated by compiler error in Figure B.2., CC2420Control interface could not be found in NetworkInterfaceM and NetworkInterfaceC components. The CC2420 is a low-power transceiver which is used in MicaZ mote, Telos motes and Tmote Sky mote as mentioned in Section 2.2.1 and 2.2.2. CC2420control interface reduces the radio power setting. However, CC2420Control interface is not necessary for Mica and Mica2 motes.

In order to solve these compilation errors, a two-step fix should be applied. First, the user should find the "NetworkInterfaceM.nc" file in /opt/tinyos-1.x/contrib/wustl/apps/Agilla/Components/NetworkInterface directory. Then, user should delete the following lines:

"interface C2420Control;"

"call CC2420Control.SetRFPower(AGILLA_RF_POWER);"

In the second step, the user should find "NetworkInterfaceM.nc" file in the same directory. Then, find the line containing "components MessageBufferM, CC2420RadioC;" and delete the word "CC2420RadioC". The user should also delete the following line completely from this file:

“NIM.CC2420Control -> CC2420RadioC.CC2420Control;”

```
…
In file included from components/NetworkInterface/NetworkInterfaceC.nc:54,
from components/NetworkInterface/NetworkInterfaceProxy.nc:56,
from components/AddressMgrC.nc:52,
from components/AgillaEngineC.nc:142,
from Agilla.nc:80:
In component `NetworkInterfaceM':
components/NetworkInterface/NetworkInterfaceM.nc:75:
interface CC2420Control not found
components/NetworkInterface/NetworkInterfaceM.nc: In function
`StdControl.init':
components/NetworkInterface/NetworkInterfaceM.nc:144:
interface has no command or event named `SetRFPower'
In file included from
components/NetworkInterface/NetworkInterfaceProxy.nc:56,
from components/AddressMgrC.nc:52,
from components/AgillaEngineC.nc:142,
from Agilla.nc:80:
In component `NetworkInterfaceC':
components/NetworkInterface/NetworkInterfaceC.nc: At top level:
components/NetworkInterface/NetworkInterfaceC.nc:55:
component CC2420RadioC not found
components/NetworkInterface/NetworkInterfaceC.nc:80:
cannot find CC2420Control'
```

Figure B.2. Compiler error of CC2420Control bug

LQI (Link Quality Indicator) Bug

The error which can be seen in Figure B.3 prevents successful compilation of Agilla. Deletion of the "nbrs[indx].linkQuality = m->lqi" (line 445 in NeighborListM.nc) solves the error.

```
components/ContextDiscovery/NeighborListM.nc: In function
`RcvBeacon.receive':
components/ContextDiscovery/NeighborListM.nc:445: structure has no member
named `lqi'
```

Figure B.3. Compiler error of LQI bug

APPENDIX C. SETTING UP THE SIMULATION ENVIRONMENT

Simulation Parameters

- Network Interface Receive Queue Size: The maximum number of incoming messages Agilla should enqueue. A higher value increases reliability of the communication and memory usage. Value: 3
- Network Interface Send Queue Size: The maximum number of outgoing messages Agilla should enqueue. A higher value increases reliability of the communication and memory usage. Value: 3
- Sender Retry Timer: The amount of time before the agent sender aborts the migration process and retries from the beginning. Value: 512 (milliseconds)
- Sender Maximum Retries: The maximum number of times the agent sender will retry migrating an agent before permanently aborting of the migration. Value: 2
- Sender Retransmit Timer: The amount of time before the agent sender retransmits a message. The message is not retransmitted if an acknowledgement is received within this. Value: 256 (milliseconds)
- Sender Maximum Retransmits: The maximum number of times the agent sender will retransmit a message before aborting of the migration. Value: 4
- Sender Abort Timer: The amount of time the sender pauses before aborting an agent in order to guarantee that the destination node times out the receive process and releases the memory. Value: 1536 (milliseconds)
- Maximum Number of Neighbors: The size of the neighbor list. Value: 20

- Remote Tuple Space (RTS) Timeout: The maximum amount of time a node waits for an acknowledgement when performing a RTS operation. Value: 256 (milliseconds)

- RTS Maximum Number of Tries: The maximum number of times a node will retry transmitting a tuple space operation before trimming out. Value: 3

Prerequisite Software

Some software must be installed before setting up the simulation environment.

Sun's JDK version 1.4.x and Sun's javax.comm package must be installed before installing TinyOS on both Windows and Linux. There are newer versions of JDK. However, some TinyOS Java programs cannot be compiled with JDK 1.5+.

In addition to Sun’s software, Graphviz must be installed to visualize some TinyOS applications. Graphviz is an open source graph visualization tool (Elson, Gansner, Koutsofios, North, & Woodhull, 2001)

Installing TinyOS

TinyOS must be installed on a PC to build up the simulation environment. A notebook PC which has an Intel Core 2 Duo processor is used for this study.

TinyOS can be installed on a Linux operating system without any auxiliary software like an emulator. On the other hand, it can be installed on a Microsoft Windows operating system with the support of Cygwin. Cygwin is a collection of tools which allows to be emulated on Microsoft Windows.

In early experiments, we installed TinyOS version 1.1.15 on the Windows XP operating system with the assistance of Cygwin version 1.5.24. We successfully installed Cygwin on Windows XP and TinyOS on Cygwin. However, we frequently met problems in the Windows environment.

We, therefore, preferred to install TinyOS on a Linux operating system to create a more stable simulation environment.

Ubuntu operating system version 6.06 was installed on the notebook PC. Ubuntu is an easy to use and flexible Linux operating system (Hill, Bacon, Burger, Jesse & Krstic, 2006). After installing Ubuntu, TinyOS version 1.1.15 was installed from TinyOS repository of Stanford University.

Installing Agilla

Agilla middleware can be downloaded and installed after a successful TinyOS installation. A user can download Agilla in two ways: through CVS (concurrent versions system) version and through pre-packaged zip file. We used pre-packaged and compressed file of the latest version (3.0.3.) of Agilla. After downloading the compressed file, we extracted it to */opt/tinyos-1.x* directory.

After extracting the zip file to the appropriate directory, it was located in three subdirectories:

- Directory which consists of the Agilla firmware:

  /opt/tinyos-1.x/contrib/wustl/apps/Agilla:

- Directory which consists of Java applications of Agilla

  /opt/tinyos-1.x/contrib/wustl/tools/java:

- Directory which consists of sample mobile agents

  /opt/tinyos-1.x/contrib/wustl/apps/AgillaAgents:

The last step of Agilla installation is compiling firmware and Java parts. The *make* is used to compile these parts. Specifically;

- *"make /opt/tinyos-1.x/contrib/wustl/tools/java/edu/wustl/mobilab/agilla"* command is used to compile the Java part, and

- *"make /opt/tinyos-1.x/contrib/wustl/apps/Agilla/"* command is used to compile the firmware part.

Compiling TinyViz

The next step of constructing the simulation environment is compiling the TinyViz application. TinyViz is embedded in TinyOS and we have already downloaded it with TinyOS. However, TinyViz script must be compiled to run. The following command is used to compile TinyViz:

"make /opt/tinyos-1.x/tools/java/net/tinyos/sim"

Installing the Agilla Simulator

The final step of setting up the simulation environment is compiling the Agilla Simulator. Presently, Agilla Simulator must be manually installed in the */opt/tinyos-1.x/contrib/wustl/tools/java/edu/wustl/mobilab/agilla/simulator* directory. However, main developer of Agilla (C. L. Fok) offered to include our Agilla Simulator application in the new Agilla distributions.

The following command is used to compile the Agilla Simulator application:

make /opt/tinyos-1.x/contrib/wustl/tools/java/edu/wustl/mobilab/agilla/simulator