

**QUERY DISSEMINATION AND PROCESSING IN
WIRELESS SENSOR NETWORKS**

**M.Sc. Thesis by
Ömer Sinan KAYA**

Department : Computer Engineering

Programme: Computer Engineering

**Supervisor : Prof. Dr. Bülent ÖRENCİK
Coadvisor : Prof. Dr. Şebnem BAYDERE**

APRIL 2004

**QUERY DISSEMINATION AND PROCESSING IN
WIRELESS SENSOR NETWORKS**

**M.Sc. Thesis by
Ömer Sinan KAYA, B.Sc.**

(504001571)

Date of submission : 26 April 2004

Date of defence examination: 21 May 2004

Supervisor (Chairman): Prof. Bülent ÖRENCİK (İ.T.Ü.)

Coadvisor: Prof. Şebnem BAYDERE (Y.Ü.)

Members of the Examining Committee Prof. Emre HARMANCI (İ.T.Ü.)

Assoc. Prof. Erdal ÇAYIRCI (W.C.)

Asst. Prof. Gökhan Yavuz (Y.T.Ü.)

MAY 2004

**TELSİZ DUYARGA AĞLARINDA SORGU DAĞITIMI VE
ÇÖZÜMLEMESİ**

**YÜKSEK LİSANS TEZİ
Müh. Ömer Sinan KAYA
(504001571)**

**Tezin Enstitüye Verildiği Tarih : 26 Nisan 2004
Tezin Savunulduğu Tarih : 21 Mayıs 2004**

**Tez Danışmanı : Prof.Dr. Bülent ÖRENCİK (İ.T.Ü.)
Ek Danışman : Prof.Dr. Şebnem BAYDERE (Y.Ü.)
Diğer Jüri Üyeleri Prof.Dr. Emre HARMANCI (İ.T.Ü.)
Doç.Dr. Erdal ÇAYIRCI (H.A.)
Yrd.Doç.Dr. Gökhan YAVUZ (Y.T.Ü.)**

MAYIS 2004

ACKNOWLEDGEMENTS

I would like to express my deep appreciation to Professor Bülent ÖRENCİK and Professor Şebnem Baydere for their guidance and tolerance while I was working for my thesis.

I am grateful to TNL members of Yeditepe University, Mesut Ali Ergin, Özlem Durmaz, Sinan Buyruk, Metin Koç and Onur Ergin for their all helps and for the time they spent to answer my questions patiently, especially to Metin Koç for patient testing and verification of the code and to Professor Şebnem Baydere for instant guidance during the project and knowledge sharing.

I thank to all people in TUBITAK-UEKAE for their continuous support for my thesis and for the experience I got during my work which helped my thesis with reasonable support.

And finally I would like to express all my gratitude to my family.

April 2004

ÖMER SİNAN KAYA

TABLE OF CONTENTS

ABBREVIATIONS	v
LIST OF TABLES	vii
LIST OF FIGURES	viii
ÖZET	x
SUMMARY	xi
1 INTRODUCTION	1
2 WIRELESS COMMUNICATION	3
2.1 Wireless Channel	4
2.2 Multiple Access Control	5
2.2.1 CSMA/CA	5
2.2.2 Hidden Terminal Problem	6
2.2.3 Exposed Terminal Problem	7
2.3 Ad Hoc Networks	7
2.3.1 Wireless Mobile Ad Hoc Networks	8
2.3.2 Wireless Ad Hoc Backbone Networks	9
2.3.3 Wireless Ad Hoc Sensor Networks	10
2.3.3.1 MEMS	11
2.3.3.2 Smart Dust	12
2.3.3.3 Networked Embedded Systems Technology (NEST)	14
2.3.3.4 Ad Hoc Sensing	14
2.3.3.5 Design Considerations	15
2.3.3.6 Application Taxonomy	18
3 STATE OF THE ART IN WIRELESS MICRO-SENSORS	20
3.1 Query Dissemination	20
3.1.1 SPIN	22
3.1.2 Directed Diffusion	24
3.1.3 LEACH	27
3.2 Query Processing and Resolution	29
3.2.1 TINYDB	29
3.2.2 COUGAR	37
3.2.3 ACQUIRE	40
4 SeMA QUERYING PROTOCOL FOR MICRO-SENSORS (SQS)	42
4.1 SeMA Architecture	43
4.2 Query Resolution	45
4.3 Query Definition	46
4.4 Compilation Functions	47
4.5 Query Types	47
4.6 Query Distribution	49
4.7 Query Processing	50

4.8	Response Generation	53
4.8.1	Response Traffic Analysis	55
4.8.2	Response Message and Time Synchronization	56
4.8.3	Response Generation Method	59
4.8.3.1	Immediate Message Delivery	59
4.8.3.2	Aggregated Message Delivery	59
4.8.4	Response Delivery	62
5	SOFTWARE ARCHITECTURE	63
5.1	Development Platform	63
5.1.1	Sensor Mote Platform	63
5.1.2	TINYOS	65
5.1.2.1	TinyOS Execution Model: Event Based Execution	66
5.1.2.2	TinyOS Component Model	67
5.1.2.3	AM Communication Paradigm	69
5.1.3	NesC	70
5.1.3.1	Component Specification	71
5.1.3.2	Component Implementation	73
5.1.3.3	Concurrency and Atomicity	73
5.1.4	SerialForwarder	75
5.1.5	TOSSIM	76
5.1.5.1	Network Monitoring and Packet Injection	79
5.1.5.2	Radio Models	79
5.1.5.3	ADC Models	80
5.1.5.4	TinyViz	81
5.1.5.5	Concurrency Model	82
5.2	Application Framework	82
5.2.1	SeMA Sensor Protocol Application	84
5.2.2	Driver Application	90
5.2.3	Querier Application	90
5.2.4	Oscilloscope Application	91
5.2.5	External Data Binding Mechanisms	92
6	EXPERIMENTS AND TESTING	94
6.1	Packet Level Data Transmission Add-on for TOSSIM	94
6.2	Power Management Add-on for TOSSIM	96
6.3	Performance and Functionality Tests	97
6.4	Timing Tests	102
6.5	Data Traffic Analysis	105
7	CONCLUSION	111
	REFERENCES	114
	APPENDIX A. EMSTAR	119
	APPENDIX B. IN NETWORK PROGRAMMING	122
	AUTOBIOGRAPHY	124

ABBREVIATIONS

ACQUIRE	: Active Query Forwarding in Sensor Networks
ADC	: Analog To Digital Converter
AM	: Active Message
AODV	: Ad hoc On Demand Distance Vector Routing Protocol
avg	: Average
BCN	: Backbone Capable Node
BER	: Bit Error Rate
BN	: Backbone Network
COTS	: Commercial Off The Shelf
CDMA	: Code Division Multiple Access
CSMA	: Carrier Sense Multiple Access
CSMA/CA	: Carrier Sense Multiple Access with Collision Avoidance
CSMA/CD	: Carrier Sense Multiple Access with Collision Detection
CRC	: Cyclic Redundancy Check
CTS	: Clear to Send
DARPA	: Defense Advanced Research Projects Agency
DIFS	: Distributed Interframe Space
DSN	: Data Source Name
ERS	: Expanding Ring Search
fd	: Flooding Degree
FDMA	: Frequency Division Multiple Access
GUI	: Graphical User Interface
hc	: Hop Count
LAN	: Local Area Network
LEACH	: Low Energy Adaptive Clustering Hierarchy
MAC	: Medium Access Layer
MANET	: Mobile Ad Hoc Network
Max	: Maximum
MBN	: Mobile Backbone Network
MEMS	: Micro-Electro-Mechanical System
Min	: Minimum
nc	: Neighbor Count
NesC	: Network Embedded Systems C
MoTeS	: Monitoring Tool for Networked Sensors
NEST	: Network Embedded System Technology
NoCOL	: No Collision
NoPER	: No Packet Error Rate
ODBC	: Open Database Connectivity
PAN	: Personal Area Network
PER	: Packet Error Rate
PLP	: Packet Loss Probability
PRN	: Packet Radio Network
QCycle	: Query Cycle

QTP	: Query Timeout Period
RF	: Radio Frequency
RFM	: Radio Frequency Model
RGP	: Response Generation Period
RN	: Regular Node
RTS	: Request to Send
S	: Sampling Period
SC	: Sample Count
SeMA	: Session Based Multi-Layer Ad Hoc Network Architecture
SHM	: Structured Health Monitoring
SPIN	: Sensor Protocols for Information via Negotiation
SQS	: SEMA Querying Protocol For Micro-Sensors
TASK	: Tiny Application Sensor Kit
TDMA	: Time Division Multiple Access
TinyOS	: Tiny Micro threading Operating System
TOSSIM	: Tiny Micro threading Operating System Simulator
WSN	: Wireless Sensor Network
XML	: eXtensible Markup Language

LIST OF TABLES

	<u>Page Number</u>
Table 2.1 Smart Dust Prototype Nodes	13
Table 4.1 Time unit encoding	51
Table 4.2 Sub query Context Encoding	52
Table 4.3 Compilation Function Encodings	52
Table 4.4 Sample Queries	53
Table 4.5 Traffic Intensity in N-hop sensor network	56
Table 4.6 Number of Sub queries vs. Concatenated Message Count	61
Table 5.1 The Energy Dissipation of Operations for Mica2	64
Table 6.1 Simulation Parameters	97
Table 6.2 Query Parameters Used For Simulation	98
Table 6.3 Timing Test Simulation Parameters	102
Table 6.4 Local Processing Test Query Parameters	106
Table 6.5 Connectivity Ratio for Data Traffic Simulation	107

LIST OF FIGURES

	<u>Page Number</u>
Figure 2.1 Cellular System.....	3
Figure 2.2 Propagation Mechanisms.....	4
Figure 2.3 CSMA/CA	6
Figure 2.4 Hidden Terminal Problem	7
Figure 2.5 Exposed Terminal Problem	7
Figure 2.6 Mobile Ad Hoc Network	8
Figure 2.7 Backbone Network	10
Figure 2.8 Smart Dust Mote Design.....	12
Figure 2.9 Ad Hoc Sensing	15
Figure 3.1 Classic Flooding	21
Figure 3.2 Packet Implosion	21
Figure 3.3 SPIN Data Exchange	24
Figure 3.4 A Sample interest description	25
Figure 3.5 Interest dissemination	26
Figure 3.6 LEACH Cluster Formation.....	28
Figure 3.7 TinyDB GUI.....	31
Figure 3.8 Triggering SQL query.....	31
Figure 3.9 A Sample TAG Query	32
Figure 3.10 Partitioning of time into EPOCHS	35
Figure 3.11 The TASK Architecture.....	36
Figure 3.12 Query Template	37
Figure 3.13 A Cougar Query Plan.....	39
Figure 3.14 ACQUIRE Query Resolution	40
Figure 4.1 SeMA Architecture	44
Figure 4.2 XML definition of a monitoring service.....	45
Figure 4.3 Query Driven Network Types	49
Figure 4.4 Setup Packet.....	50
Figure 4.5 Time Unit Structure	51
Figure 4.6 Sampling Period Timer Algorithm	54
Figure 4.7 RGP Periods Time Gantt chart: (SC*S) per sub query.....	55
Figure 4.8 Response Packet	57
Figure 4.9 Response Data	57
Figure 4.10 Timing Problem Demonstration	58
Figure 4.11 Response Packets in Detail	60
Figure 4.12 Concatenation Algorithm.....	61
Figure 5.1 COTS Motes	64
Figure 5.2 Mote Programming Interface.....	65
Figure 5.3 TinyOS Execution Model	66
Figure 5.4 Component Graph of GenericComm.....	67
Figure 5.5 A sample component: Messaging Component.	68
Figure 5.6 TinyOS Packet Format.....	69
Figure 5.7 Timer Module Configuration.....	72

Figure 5.8	Relationship diagram of Timer Module.....	72
Figure 5.9	Sample Interfaces.....	73
Figure 5.10	Atomic Usage	74
Figure 5.11	SerialForwarder	75
Figure 5.12	TOSSIM Architecture.....	76
Figure 5.13	Flow Diagram of TOSSIM	77
Figure 5.14	A Sample Run of TOSSIM.....	78
Figure 5.15	Lossy Model Configuration	80
Figure 5.16	TinyViz	81
Figure 5.17	SQS Application Framework.....	83
Figure 5.18	Hardware Dependent Definitions	84
Figure 5.19	SeMA Querying Application Component Graph	86
Figure 5.20	BitParser Interface	87
Figure 5.21	MsgForwardBuffer Interface	88
Figure 5.22	ADCBuffer Interface	89
Figure 5.23	Querier Application	91
Figure 5.24	Oscilloscope Application.....	92
Figure 5.25	Database Design	93
Figure 5.26	ODBC Access of Sensor Data from The Database.....	93
Figure 6.1	Packet Level Transmission Configuration.....	95
Figure 6.2	Connectivity for Immediate Delivery	98
Figure 6.3	Immediate Event Delivery	99
Figure 6.4	Event Delivery for Aggregate Event Delivery	100
Figure 6.5	Message Delivery Ratio for Immediate Delivery	100
Figure 6.6	Message Delivery Ratio for Aggregate Message Delivery	101
Figure 6.7	Timing Test Topology	103
Figure 6.8	Flooding Degree vs. Average Response Time.....	103
Figure 6.9	Flooding Degree vs. Message Count	104
Figure 6.10	The Best and Worst Cases	105
Figure 6.11	Logical Topology for Data Traffic Simulation.....	108
Figure 6.12	Data Traffic Analysis.....	109
Figure A.1	EmStar Software Stack	120
Figure A.2	EmStar Simulator Architecture.....	120
Figure B.1	Network Programming.....	122
Figure B.2	Xnp Network Programming Protocol	122

ÖZET

Telsiz ağlar bilginin kolay erişilir olmasını sağladı. Çevre ile etkileşebilme yeteneği ile birleştirildiğinde, gelecekte her yerde işlem yapabilme (ubiquitous computing) günlük hayatın bir parçası olacak. Smart DUST ve diğer DARPA destekli projelerin tamamlanması ve MEMS teknolojisindeki yenilikler ile duyarga düğümleri bir milimetre küp hacme sahip olacaklar ve duyarga ağlarının maliyetinin 1\$ civarında olması bekleniyor. Bu teknolojik gelişmeler yoğun duyarga ağlarını kullanıma açacak. Telsiz duyarga ağları uzak yerlerin insane müdahalesi olmadan gözlemlenebilmesini sağlar. Bu ağlar müdahalesiz ve bağlantısız bir şekilde çalışır ve güç harcaması çok yüksek öneme sahiptir. Genel amaçlı kendiliğinden uyum sağlayabilen altyapılar kullanıcı isteklerinin dağıtımı ve duyarga verilerinin toplanması için kullanışlıdır. Bu tezde gücü verimli bir şekilde kullanmak için bir yerel sorgu optimizasyonu yapan ve ağ içinde mesajları bitişiren bir duyarga sorgulama mekanizması tanıtılmaktadır. Protokol bir çok katmanlı mimarinin en alt katmanı olarak çalışması üzere tasarlanmıştır. Bu mimaride protokol bir gözlem uygulaması aracılığıyla hizmet tabanlı bir tasarısız omurga üzerinden hedef alandaki duyargalara erişimi sağlamaktadır. Tezde sorgular veri üretme karakterlerine göre kıyaslanmaktadır ve bu bağlamda sorgulama sisteminin bileşenleri açıklanmaktadır. TinyOS üzerinde çalışan Mica düğümleri için bir prototip gerçekleştirme yapılmıştır ve bu gerçeklemenin deneysel performans sonuçları ve karşılaştırmalı veri trafiği verilmiştir. Sonuçlar göstermiştir ki yerel optimizasyon ile veri trafiği kullanıcının belirlediği örnekleme periyodu oranında azalır ve mesaj bitleştirme ise ayrıca veri iletimini %75 oranında azaltır. Ayrıca yine gösterilmiştir ki uygulamaya özel hizmet kalitesi parametreleri dinamik duyarga ağlarının oluşturulması için verimli bir şekilde kullanılabilir.

SUMMARY

Wireless networking has enabled easy access of information. Combined with the power of interacting with the environment, ubiquitous computing will be a part of modern day life in the future. With the completion of Smart Dust and other DARPA supported projects and improvement in MEMS technology, sensor nodes will have a size of cubic millimeter and sensor node cost is expected to be around 1\$. These technological advancements will enable dense sensor networks to be available for use. Wireless sensor networks enable inspection of remote locations without any human intervention. They operate in unattended and untethered mode where power consumption is of greatest importance. General purpose adaptive infrastructures are useful for dissemination of user interests and collection of sensor data. In this thesis, a sensor querying mechanism which provides local query optimization and in-network message concatenation for energy efficient response delivery is presented. The protocol is designed to operate as the lowest tier of a hierarchical architecture that enables access to tiny sensors in the target domain from a monitoring application via a service aware ad hoc backbone. Queries are classified according to data generation characteristics and the components of the querying system are explained in this context. A comparative data traffic analysis and experimental performance results of the prototype implementation on TinyOS running Berkeley mica motes are also given. The results have revealed that local optimization can reduce bit transmission by a multiple of user supplied sampling period parameter and message concatenation can further reduce transmission by 75%. It has also been shown the application specific quality of service parameters can effectively be used to construct sensor querying networks dynamically.

1 INTRODUCTION

With the emerging Micro-Electro-Mechanical System (MEMS) technologies sensor nodes are able to combine power processing, data transmission and sensing capabilities into one small node that operates with battery or other sources of energy such as solar power. It's expected that in the near future, sensor nodes will have a dimension of 1 cubic millimeter. As a result of this decrease in size, large number of sensor nodes will be able to be used in applications to disseminate and process data in a distributed fashion. While the capabilities of these nodes are limited, in order to handle large number of data flowing from numerous nodes in a power efficient way, new technologies are required. The power of wireless sensor networks lies in the ability to deploy large number of sensor nodes that are able to assemble and configure themselves resulting in adaptive systems.

The most straightforward application of wireless sensor networks is to monitor a remote location in environment to receive low frequency responses. For example, a forest can be monitored against fire and sensor nodes can report temperature every 5 minutes. With a well designed protocol, a sensor node can serve for one year for requested monitoring capability until it runs out of battery. Since nodes are distributed to monitoring terrain in a random fashion, it's not possible to access the nodes again.

Most of the studies aim to problems in Medium Access Layer (MAC) or other layers. However, sensor networks are formed by an application. Therefore, it is important to know the traffic characteristics of the applications and adopt the system according to application requirements. Querying is the application layer of sensor network architecture. Therefore, a set of options are provided to user by querying layer to fit the requirements of different kinds of applications. In this thesis, a general purpose scalable and power efficient query dissemination and processing system used as the lowest tier of a multi-tier architecture for processing user data requests from sensor networks for environmental monitoring is proposed.

In network programming schemes could also be used for small sensor network scenarios as a replacement of querying systems. The difficulty behind in network programming is that distribution of code updates and maintaining synchronized software updates is a cumbersome task. Therefore, general purpose sensor network applications that can fit most of the user requests are a must for dense, scalable sensor networks.

Contents of this thesis are arranged as below:

In Chapter 2, basics of wireless communication are discussed and the subject is further enhanced by the description and limitations of wireless ad hoc sensor networks.

In Chapter 3, what is meant by query dissemination and processing is defined and the current state of the art in query processing and dissemination protocols is given.

In Chapter 4, Session Based Multi-Layer Ad Hoc Network Architecture (SeMA) is described and how SeMA sensor querying protocol for micro-sensors fits into the architecture is shown and the protocol details are given.

In Chapter 5, sensor network development platform is introduced and software architecture is given in detail.

In Chapter 6, results of several experiments and tests are given.

In Chapter 7, conclusion and comparison of SEMA Querying Protocol for Micro-Sensors (SQS) with other query processing protocols are given.

In Appendix A, brief description of EmStar development environment which is a scalable and robust simulator that enables mixed mode operation of sensor nodes and simulated nodes is presented.

In Appendix B, introductory information about Crossbow [32] implementation of an in network programming scheme is given.

2 WIRELESS COMMUNICATION

Hardware connections and electronic switches have made transfer of digital data feasible over long distances. There is a long history of how the field has evolved [1]. The use of internet has added another dimension to the wireline communication field, and both voice and data are being processed extensively. In parallel to wireline communication, radio transmission has progressed substantially. Feasibility of wireless transmission has brought drastic changes in the way people live and communicate.

Wireless communication has become popular by the introduction of affordable wireless telephones to the public society. Wireless systems have evolved over time by the beginning of first-generation (1G) towards third generation telephone systems. Later, wireless communication has become very popular in major fields such as commerce, medicine, education and military defense. The ease of access to data made this technology more demanding.

Cellular systems have been in use for a long time and the idea of forming an ad hoc network takes most of its requirements from the cellular systems limitations. In a cellular system a base station serves to mobile nodes in the range of a base station's transmission range as in depicted in Figure 2.1.

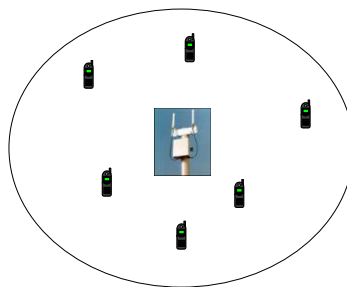


Figure 2.1 Cellular System

The radius of the circle is equal to the reachable range of the transmitted signal. In each cell, multiple users or wireless subscribers are served by a single base station. If the coverage area is to be increased, then additional base stations are placed to take care of the added area. Time Division Multiple Access (TDMA), Frequency Division

Multiple Access (FDMA) or Code Division Multiple Access (CDMA) is used to distribute available bandwidth among the users. The idea behind cellular systems is to provide users with mobile telephony service over a fixed base station to base station network infrastructure.

2.1 Wireless Channel

For wireless and mobile system design, it is very important to understand the distinguishing features of mobile radio propagation. There are several kinds of radio waves, such as ground, space, sky and satellite waves. We'll refer to ground and space waves in this section.

Propagation in free space and without any obstacle is the most ideal situation. When the radio waves reach close to an obstacle, the following propagation effects as in Figure 2.2 do occur to waves:

- **Reflection:** Propagating wave hits an object such as tall buildings, surface of the earth that is larger as compared to its wavelength.
- **Diffraction:** Radio path between a transmitter and a receiver is obstructed by a surface with sharp irregular edges.
- **Scattering:** When objects are smaller than the wavelength of the propagation wave, the incoming signal is scattered into several weaker outgoing signals.

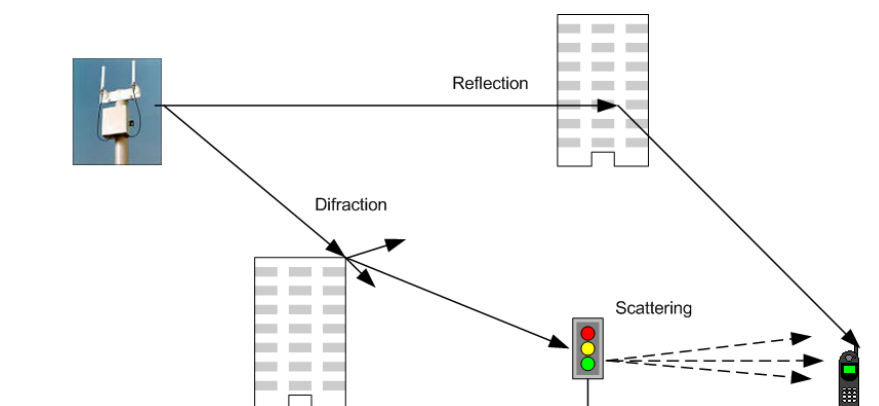


Figure 2.2 Propagation Mechanisms

Free space is the ideal transmission medium for wireless communication. At an arbitrary, large distance d from the source, the radiated power is uniformly distributed over the surface area of a sphere of radius. Thus, the received signal power varies inversely with the square of d .

However, not all of the communication is achievable using free space communication. Signal propagates through the land to reach the target user as well. In the case of land propagation, radio channel becomes a multi path propagation channel with fading. The signal reaches the destination using many different paths because of the diffraction and reflection from various objects along the path. The signal strength and quality of received radio waves also vary accordingly as well as the time to reach the destination changes.

Wave propagation in a mobile radio channel is characterized by three aspects: path loss, slow fading (shadowing) and fast fading. The path loss is the average propagation loss over a wide area. It is determined by the macroscopic parameters, such as the distance between the transmitter and receiver, the carrier frequency and the land profile. The slow fading loss represents variation of the propagation loss in a local area. Slow fading is caused by the variation in propagation conditions due to buildings, roads and other obstacles in a relatively small area. Fast fading is due to the motion of the terminal in a standing wave that consists of many diffracted waves representing the microscopic aspect of the channel.

2.2 Multiple Access Control

Existing Local Area Networks (LANs), Packet Radio Network (PRNs) and Personal Area Networks (PANs) do utilize broadcast channels rather than point-to-point channels for information transmission. MAC sub layer protocols are primarily a set of rules that communicating terminals need to follow, and these are assumed to be agreed upon a priori. MAC layers are classified as contention based or conflict free. Conflict free protocols such as FDMA or TDMA require a centralized coordination among the communicating nodes to share the channel between participants and this kind of protocols are prone to mobility in the environment. Contention based protocols such as Carrier Sense Multiple Access (CSMA) and its variants are generally used in wireless communication.

2.2.1 CSMA/CA

CSMA protocols try to increase throughput by listening to channel before transmitting data for some time to see if another node is also transmitting and they keep their state information locally which makes CSMA protocols most suitable for wireless communication.

Carrier Sense Multiple Access with Collision Detection (CSMA/CD) used in Ethernet does not fit into wireless networking. Because carrier sensing is much costlier in wireless and due to radio hardware the transmitter node can not listen while transmitting. As a result, a node can not abort a transmission by detecting a collision as in the case of CSMA/CD. Additionally, in wireless systems, collisions occur at the receiver and due to spatial distribution a transmitter can not listen to the channel at the receiver.

Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) as defined by IEEE 802.11 is a modified version of CSMA/CD protocol. Details of CSMA/CD can be found in [2]. Under a CSMA/CA technique, all terminals listen to the medium same as CSMA/CD. A terminal that is ready to transmit data senses the medium and will transmit its data if the medium is idle for time interval that exceeds the Distributed Interframe Space (DIFS); otherwise, if the medium is busy, it waits for an additional predetermined time period, denoted as DIFS, and then picks a random backoff period within its contention window to wait before transmitting its data. The backoff period is used to initialize the backoff counter. The backoff counter can count down only when the medium is idle. Otherwise, the node's frozen counter resumes only after the medium has been free longer than DIFS. The terminal can start transmitting its data when the backoff counter becomes zero. Collisions can occur only when two or more terminals select the same time slot in which to transmit their frames. The flow of the algorithm is shown in Figure 2.3.

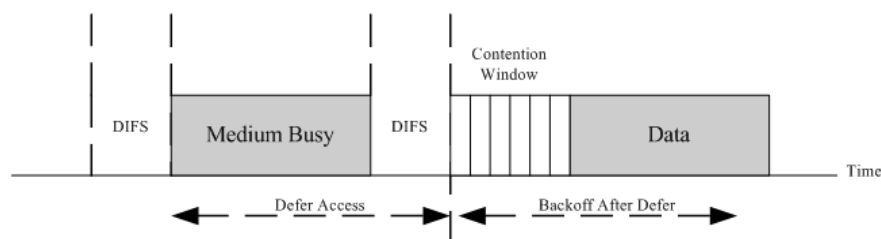


Figure 2.3 CSMA/CA

2.2.2 Hidden Terminal Problem

The transmission range of wireless radio hardware is usually limited. Hidden terminal problems occur in wireless networks when two nodes, which aren't in the transmission range of each other, transmit a message to the same node as in Figure 2.4. As a result of this two packets transmitted at the same time collide and packets are lost. Ready to Send (RTS) / Clear to Send (CTS) based mechanisms are used to

avoid hidden terminal problem for unicast traffic. Protocols should avoid assuming reliable transmission in wireless networks.

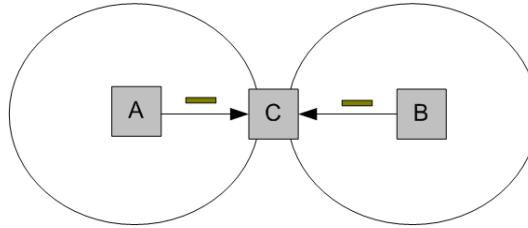


Figure 2.4 Hidden Terminal Problem

2.2.3 Exposed Terminal Problem

As stated before CSMA/CA listens for a fixed period prior to starting data transmission and delays packet transmission if there is an active transmission. In the case of Figure 2.5, A is transmitting a message to B and C tries to transmit a message to D. However, as a result of CSMA/CA, C will sense the channel as busy and will not transmit its message to D which is not in the transmission range of A and B. Consequently, the throughput of the system will decrease. Directed antennas are offered in the literature as a solution to this problem.

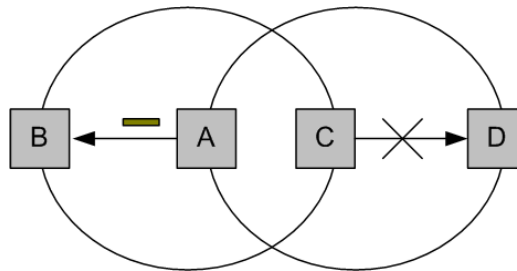


Figure 2.5 Exposed Terminal Problem

2.3 Ad Hoc Networks

Ad hoc networks are designed to remove cellular system's infrastructure requirement. An ad hoc network is a local network with wireless or temporary plug-in connection, in which mobile or portable devices are part of the network only while they are in close proximity.

An ad hoc network consists of mobile platforms, known as nodes, which are free to move around arbitrarily. Each node is equipped with a wireless transmitter and a receiver with appropriate antenna, which may be omni directional or highly directional. At a given point in time, depending on the nodes' positions and their transmitter and receiver coverage patterns, transmission power level, and cochannel

interference levels, a wireless connectivity in the form of random, multi hop graph or ad hoc network exists between the nodes. This ad hoc topology may change with time as the nodes move or adjust their transmissions and reception parameters.

There are three kinds of ad hoc networks: Mobile ad hoc networks, backbone ad hoc networks and sensor ad hoc networks.

2.3.1 Wireless Mobile Ad Hoc Networks

A Mobile Ad Hoc Network (MANET) is formed by the mobile devices that comes together to form a network as needed, not necessarily with any existing infrastructure or any other kinds of fixed stations. In MANET, the network infrastructure may change dynamically in an unpredictable manner since nodes are free to move.

Initially, the technology was developed keeping in mind the military applications of such a technology in areas such as the battlefield, where an infrastructured network is almost impossible to have or to maintain. In such situations, ad hoc networks, with their self-organizing capability, can be used effectively, where other technologies fail.

Ease and speed of deployment, personal area networking and decreased dependence on infrastructure make MANET's popular. A user can easily create a network while on the plane with a colleague or in the case of a disaster, networks can be set up very quickly without requiring a previously set up infrastructure. Computers with short-range interactions such as ear phone, wrist watch, and smart office makes life more comfortable. Intelligent devices can be connected with one another via wireless links and newly joined nodes can request information from local servers without any human intervention.

Ad hoc networks are basically peer-to-peer, multi-hop mobile wireless networks in which information packets are transmitted in a store-and-forward manner from a source to an arbitrary destination as in Figure 2.6.

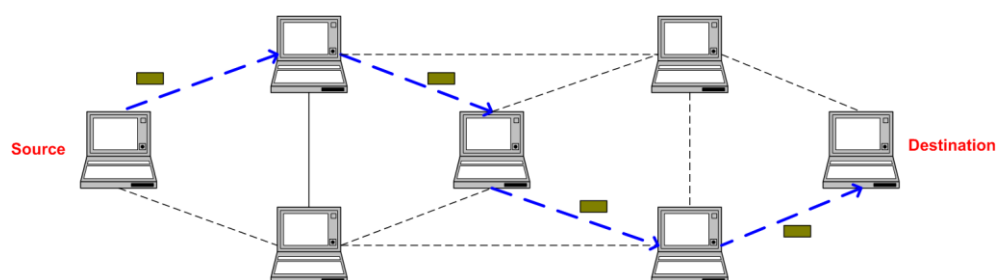


Figure 2.6 Mobile Ad Hoc Network

In an ad hoc network, each node is expected to serve as a router and each router is indistinguishable from another in the sense that all routers execute the same routing algorithm to compute routing paths through the entire network. As the number of nodes become larger, the overhead in computing, storing, and communicating the router table information could become prohibitive. Therefore, instead of a flat architecture, a hierarchical structure, with a leader of a cluster of nodes, could serve as a router to other clusters of the network.

Characteristics of Ad Hoc Networks:

Dynamic Topologies: Nodes are free to move arbitrarily; thus, the network topology may change randomly and at unpredictable times and may primarily consist of bidirectional links. In some cases, where the transmission power of two mobile nodes is different, a unidirectional link may exist.

Bandwidth-constrained and variable capacity links: Wireless links continue to have significantly lower capacity than the infrastructured networks. In addition, the realized throughput of wireless communications after accounting for the effects of multiple access, fading, noise, interference conditions and so on - is often much less than a radio's maximum transmission rate. Packet losses due to transmission errors occur very frequently and since transmission range is very short and nodes can be mobile, frequent network partitions can occur.

Energy-constrained operation: Some or all of the nodes in an ad hoc network may rely on batteries. For these systems, the most important design optimization criteria may be the energy conservation.

Security: Mobile wireless networks are generally more prone to physical security threats than wireline networks and wireless transmissions can be easily snooped.

2.3.2 Wireless Ad Hoc Backbone Networks

Wireless Ad Hoc Backbone Networks are used for easy establishment of a backbone in an area. Most of the nodes in the backbone serve as an IP router and are used to forward users data requests to some IP based network such as internet. The nodes in a wireless ad hoc backbone network can be mobile and therefore they form a Mobile Backbone Network (MBN). Detailed information about a MBN can be found in [3]. An MBN consists of a backbone network, access nets and regular ad hoc networks as in Figure 2.7. Nodes are categorized as a Regular Node (RN) and a backbone capable

node (BCN) in this network. BCNs are better equipped, have higher capacities and have ability to operate at multiple power levels and employ multiple radio modules. The MBN is designed so that it involves a sufficient but not excessive number of backbone capable nodes while providing high coverage, so that high fraction of the low power nodes can access at least a single Backbone Network (BN).

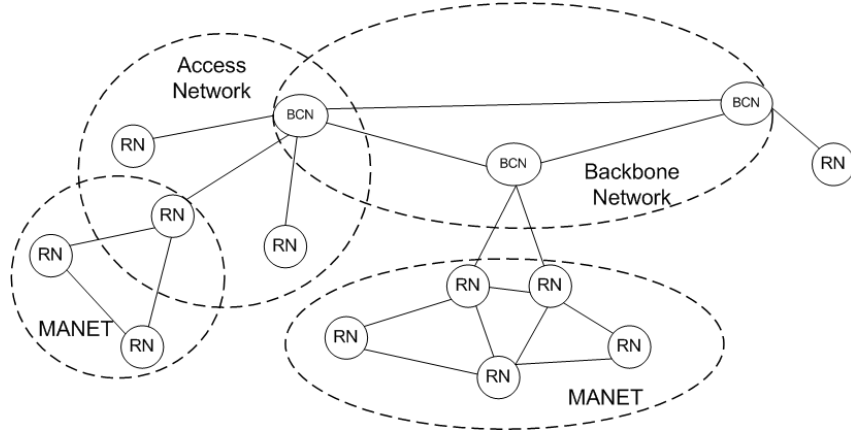


Figure 2.7 Backbone Network

Ease of deployment makes wireless ad hoc networks a promising feature in the case of a war or disaster.

2.3.3 Wireless Ad Hoc Sensor Networks

Recent advances in wireless communications and electronics (MEMS) have enabled the development of low-cost, low-power multifunctional sensor nodes that are small in size and communicate untethered in short distances [4]. A wireless sensor network is a collection of tiny disposable and low-power devices. The position of sensor nodes need not be engineered or predetermined. This allows random deployment in inaccessible terrains or disaster relief operations. A sensor node is a device that converts a sensed attribute (such as temperature) into a form understandable by users. Each of such devices may include a sensing module, a communication module, memory, CPU and a small battery. Another unique feature of sensor networks is the cooperative effort of sensor nodes. Instead of sending the raw data to the nodes responsible for fusion, they use their processing abilities to locally carry out simple computations and transmit only the required and partially processed data.

Wireless sensor networks are a new class of ad hoc networks that are expected to find increasing deployment in incoming years, as they enable reliable monitoring and analysis of unknown and untested environments [1]. These networks are data-centric

(unlike traditional networks, where data are requested from a specific node, data are requested based on certain attributes such as “which area has temperature 30°C?”). The routing protocols proposed for all the traditional networks are point to point, and these protocols are not well suited for wireless networks.

2.3.3.1 MEMS

Microelectromechanical Systems (MEMS) is one of the core enabling technologies within the microsystems. MEMS technology merges the functions of compute, communicate and power together with sense, actuate and control to change completely the way people and machines interact with the physical world [5]. Using an ever-expanding set of fabrication processes and materials, MEMS will provide the advantages of small size, low-power, low-mass, low-cost and high-functionality to integrated electromechanical systems both on the micro as well as on the macro scales. Further, demands for increased performance, reliability, robustness, lifetime, maintainability and capability of military equipment of all kinds can be met by the integration of MEMS into macro devices and systems.

MEMS is based on a manufacturing technology that has had roots in microelectronics, but MEMS has gone beyond this initial set of processes as it became more intimately integrated into macro devices and systems. MEMS will be successful in all applications where size, weight and power must decrease simultaneously with functionality increases, and all while done under extreme cost pressure. Typical applications include, but are not limited to:

- Inertial measurement units for munitions, military platforms and personal navigation;
- Electromechanical signal processing;
- Distributed control of aerodynamic and hydrodynamic systems;
- Distributed sensors both for condition-based maintenance and for structural health and monitoring;
- Distributed unattended sensors both for asset tracking and for environmental/security surveillance;
- Atomic resolution data storage devices;
- Miniature analytical instruments;

- Non-invasive biomedical sensors and
- Optical fiber components and networks.

2.3.3.2 Smart Dust

Smart Dust project is the pioneer project in this area and is further enhanced into Network Embedded System Technology (NEST) project by Defense Advanced Research Projects Agency (DARPA). The goal of the Smart Dust project is to build cubic-millimeter scale sensing and communication platforms that form a distributed sensor network [6] and can monitor environmental conditions in both military and commercial applications. These networks will consist of hundreds to thousands of “dust motes” and a few interrogating transceivers. The concern of Smart Dust is the integration of micromachined sensors and communication devices with standard CMOS circuits into a low cost, low power, small volume package.

The primary constraint in the design of the Smart Dust motes is volume, which in turn puts a severe constraint on energy since there is not much room for batteries or large solar cells. The dust motes are comprised of various subsystems as shown in Figure 2.8 from different fabrication technologies. Many sensors, including temperature, pressure, and acceleration sensors, from MEMS and CMOS processes can be attached to a mote. A microprocessor handles measurement recording, data storage and system control. Laser communication interface has been chosen as communication module. A receiver circuit converts photocurrent from an incoming laser into a data stream to be used to interrogate or reconfigure the mote.

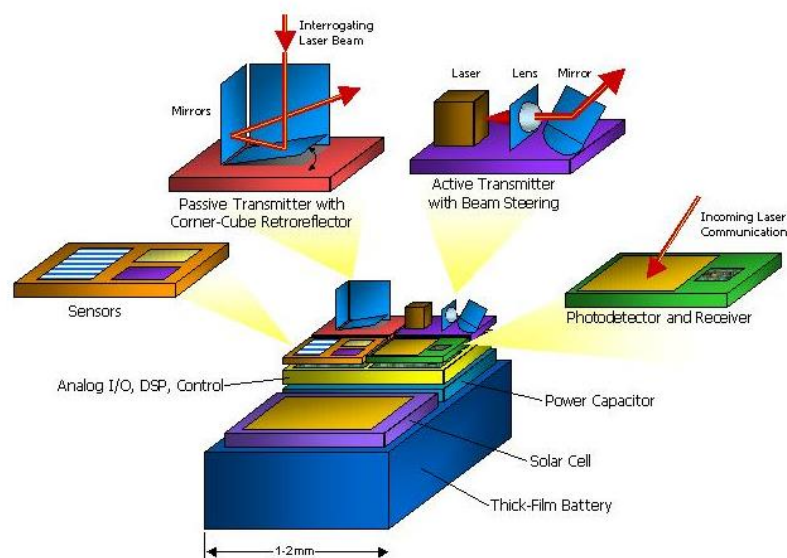



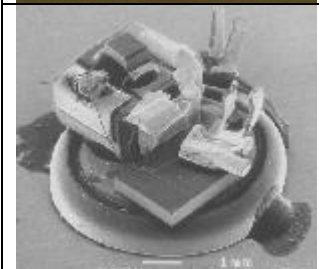
Figure 2.8 Smart Dust Mote Design

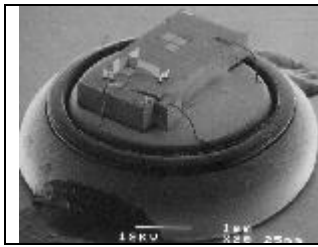
Several transmission systems can also be utilized, such as a passive corner cube reflector. Finally, all of the systems are mounted onto a thick-film battery charged with a solar cell.

The Smart Dust mote is run by a microcontroller that not only determines the tasks performed by the mote, but controls power to the various components of the system to conserve energy. Periodically the microcontroller gets a reading from one of the sensors, which measure one of a number of physical or chemical stimuli such as temperature, ambient light, vibration, acceleration, or air pressure, processes the data, and stores it in memory. It also occasionally turns on the optical receiver to see if anyone is trying to communicate with it. This communication may include new programs or messages from other motes. In response to a message or upon its own initiative the microcontroller will use the corner cube retroreflector or laser to transmit sensor data or a message to a base station or another mote.

Smart Dust project has been finished by 2001 and prototype nodes developed by this project are shown in Table 2.1. Smart Dust project has continued to develop miniature scale hardware structures that are advancing the state-of-the-art in wireless sensor network technology. In targeting extreme miniaturization and low-power consumption, many ultra-low power primitives including the radio and Analog to Digital Converter (ADC) components has been developed. Smart Dust project has pioneered new optical communication technologies through the use of MEMS mirror-based optical communication.

Table 2.1 Smart Dust Prototype Nodes

	<p>Golem Dust: solar powered mote with bi-directional communications sensing (acceleration and ambient light) 11.7 mm³ total circumscribed volume ~4.8 mm³ total displaced volume</p>
	<p>Daft Dust: X20 63 mm³ bi-directional communication mote</p>



Flashy Dust:

X12 138 mm³ uni-directional communication and sensing (ambient light) mote

2.3.3.3 Networked Embedded Systems Technology (NEST)

Networked Embedded Systems Technology (NEST) is one of the most powerful technologies that will shape science and engineering in the twenty-first century [7]. Revolutionary changes are already underway in a broad range of monitoring and control fields: transportation, manufacturing, health care, environmental oversight, virtual reality, and safety and security, to name just a few. Each of these areas has developed a deep NEST technology, with its own hardware, algorithms, mathematics, and specialized techniques.

Advances in networking and integration have enabled small, flexible, low-cost nodes that interact with their environment through sensors, actuators and communication. Single-chip systems are now emerging that integrates a low-power CPU and memory, radio or optical communication, and substantial MEMS-based onchip sensors; these nodes are casually referred to as "Motes" or "Smart Dust". Target costs (for single-chip Motes) are less than 10 cents per unit, which enables networks with potentially tens of thousands of Motes. Target power consumption means that motes can last years with low-bandwidth communication, or even be battery-free when fueled by ambient power (e.g., heat, light, or vibration from the environment).

A NEST system is a collection of autonomous motes. Each mote is networked and embedded. Networking is used to coordinate sensor nodes for collaborative sensing and perform higher level tasks. Numerous distributed devices that communicate with each other are embedded to monitor and interact with physical world.

2.3.3.4 Ad Hoc Sensing

Ad hoc sensing is the prototypical example of the use of a NEST system. It is characterized by self organizing network, sensor information and intermediate nodes as in Figure 2.9. Autonomous nodes self assemble into a network of sensors. The topology of the network is quite often a one-to-many spanning tree root at the Base Station's interface to the sensor network. Sensor information is propagated to a central collection point or "Base Station". Some applications require raw sensor

readings to be collected while other applications may be interested in aggregate function values computed over a set of sensor information. Intermediate nodes assist distant nodes to reach the Base Station by forming a Multi Hop Network.

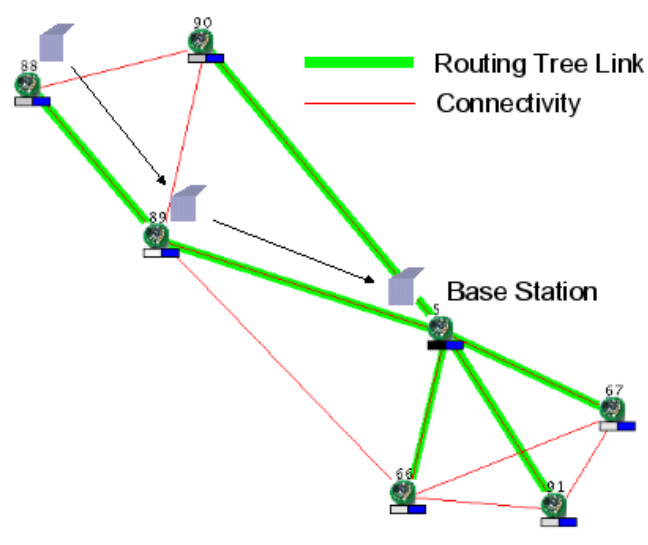


Figure 2.9 Ad Hoc Sensing

2.3.3.5 Design Considerations

Traditional protocols defined for wireless ad hoc network are not well suited for wireless sensor networks [4]. The differences between a MANETs and sensor networks are:

- The number of sensor nodes in a sensor network can be several orders of magnitude higher than the nodes in an ad hoc network.
- Sensor nodes are densely deployed.
- Sensor nodes are prone to failures.
- The topology of a sensor network changes very frequently.
- Sensor nodes mainly use a broadcast communication paradigm, whereas most ad hoc networks are based on point-to-point communications.
- Sensor nodes are limited in power, computational capacities and memory.
- Sensor nodes may not have global identification because of the large amount of overhead and large number of sensors.

Factors influencing a wireless sensor network design can be categorized as follows:

Power: There are a few inherent limitations of wireless media, such as low bandwidth, error-prone transmissions and need for collision-free channel access. Also, since the wireless nodes are mostly mobile and are not connected in any way to a constant power supply, they derive energy from a personal battery. This limits the amount of energy available to the nodes. In addition, since these sensor nodes are deployed in places where it is difficult to replace the nodes or their batteries, it is desirable to increase the longevity of the network. Thus, the protocols designed for these networks must strategically distribute the dissipation of energy, which also increase the average lifetime of the overall system.

Fault Tolerance: Since sensor networks are mostly unattended, they should possess fault-tolerant capability. Some sensor nodes may fail or be blocked due to lack of power, or have physical damage or environmental interference. The failure of sensor nodes should not affect the overall task of the sensor network. Foremost issue is to detect if a node is faulty. When a sensor crashes (either because of battery depletion or due to a catastrophic event), neighboring sensors should cover, at least partially, it's sensing task. Also, re-routing around failed nodes is necessary. Path from source to sink is set up on demand. In case of failure of nodes, alternative paths are needed. One solution is to periodically send events (network wide flooding) to find alternate paths that enable local re-routing around failed nodes. But such flooding can adversely impact the lifetime of network. It is possible to set up multiple paths just once initially.

Flexibility: The wide range of usage scenarios being considered means that the wireless sensor network must be flexible and adaptive. Each application scenario will demand a slightly different mix of lifetime, sample rate, response time and in-network processing. Wireless sensor network architecture must be flexible enough to accommodate a wide range of application behaviors. Additionally, for cost reasons each device will have only the hardware and software it actually needs for a given application. The architecture must make it easy to assemble just the right set of software and hardware components. Thus, these devices require an unusual degree of hardware and software modularity while simultaneously maintaining efficiency.

Robustness: In order to support the lifetime requirements demanded, each node must be constructed to be as robust as possible. In a typical deployment, hundreds of nodes will have to work in harmony for years. To achieve this, the system must be

constructed so that it can tolerate and adapt to individual node failure. Additionally, each node must be designed to be as robust as possible.

Scalability: Sensor nodes are expected to be dense. New schemes must be able to work with thousands of nodes.

Hardware Constraints: In order to use low power, most of the sensor nodes have limited processing capabilities and local storage. Therefore, algorithms depending on complex operations and large number of data would not be feasible for sensor networks. Transmission medium can be optical, Radio Frequency (RF) or infrared. Smart dust motes use optical communication and they require line of sight. Infrared is another communication mechanism which is robust to interference but it also requires line of sight between the transmitter and receiver. RF is generally used in academic researches which is not prone to interference and has many problems to be addressed as discussed in the Multiple Access Channel section of the thesis.

Security: Sensor Networks are dynamic in nature; they allow addition and deletion of sensor nodes after deployment to allow for growth of the network or to replace failing and unreliable nodes. An adversary may deploy sensor networks in hostile areas where communication is monitored and nodes are subject to capture and dishonest use. Hence these networks require cryptographic protection of communications, sensor capture detection, key revocation and sensor disabling. Care must also be taken to ensure that nodes are unable to act selfishly and must contribute to the routing of their neighbors' packets and the maintenance of the network. Sensor networks must also be provided authentication, integrity and privacy. At the same time, any design for sensor networks must keep in mind that sensor nodes are severely limited in terms of computational power and energy. The trust model in the network assumes that the sensor nodes are not trusted; the base station is trusted and shares keys with all nodes. It is also assumed that the broadcast medium is subject to threat.

Time Synchronization: In order to support time correlated sensor readings and low-duty cycle operation of data collection applications, nodes must be able to maintain precise time synchronization with other members of the network. Nodes need to sleep and awake together so that they can periodically communicate. Errors in the timing mechanism will create inefficiencies that result in increased duty cycles.

Thus, sensor networks need protocols that are application specific, data centric, and capable of aggregating data and minimizing energy consumption while keeping scalability and fault tolerance in design.

2.3.3.6 Application Taxonomy

There are several application areas for wireless sensor networks from military applications to commercial applications including battlefield surveillance, environment monitoring and health applications [8].

Target Tracking: Target tracking is one of the primary applications of sensor networks. The key steps involved in the tracking procedure include event detection, target classification, and estimation and prediction of target location. Tracking enemy movements in the battlefield is an example of multiple targets tracking application. There are several studies that try to determine the number of targets in the field. Generally, Collaborative signal processing schemes are used for detection and tracking of targets.

Habitat Monitoring: The remote sensing capabilities of sensor nodes allow inspection of certain species habits in their natural living conditions without any effect of human intervention. Sensor nodes work in untethered mode in an undistributed environment.

Environmental Monitoring: Environmental monitoring includes large distributed system that spans large geographic areas and monitor, model and forecast physical processes, such as environmental pollution, flooding etc.

Health Monitoring: Applications in this category include telemonitoring of human physiological data, tracking and monitoring of doctors and patients inside a hospital, drug administrator in hospitals etc. The idea of embedding wireless biomedical sensors inside human body is promising, although many additional challenges exist: the system must be ultra safe and reliable; require minimal maintenance; energy-harnessing from body heat. With more researches and progresses in this field, better quality of life can be achieved and medical cost can be reduced.

Structure Health Monitoring Systems (SHM): SHM is another important domain for sensor network application. The widely accepted goals of SHM system include detecting damage, localizing damage, estimating the extent of the damage and

predicting the residual life of the structure [9]. Wireless Sensor Network (WSN) provides low deployment and maintenance cost, large physical coverage and high spatial resolution compared to wired sensors for this case.

Home Applications, Office Applications: “Smart Kindergarten” proposed by [10] builds a sensor-based wireless network for early childhood education. It is envisioned that this interaction-based instruction method will soon take place of the traditional stimulus-responses based methods.

3 STATE OF THE ART IN WIRELESS MICRO-SENSORS

In this section, what's meant by Query Dissemination and Query Processing is explained. In each of these subsections, general methods used in existing Internet protocols or other systems for disseminating and processing data are first introduced. Later, most cited academic studies and papers in these areas are summarized.

3.1 Query Dissemination

The main purpose of a sensor network is to gather information by monitoring the environment. This is possible by collecting information from all the sensors distributed in the environment. Data dissemination protocols define methods for nodes to transmit and receive queries and sensing data in wireless sensor networks efficiently [8]. Thus, data dissemination protocols help sensor networks achieve their aim of gathering information from multiple nodes.

In order to collect information from all the nodes, some level of routing of data must be performed by the data dissemination protocols. Also, when a large number of sensors sense the same phenomena, each node, which sensed the phenomena, may propagate information on this phenomenon and may hence send all other nodes, which already have information on this phenomenon. This kind of propagation of information causes network congestion.

Another distinctive attribute of a sensor network is that all communication is data centric. In wireless sensor networks, nodes cannot designate the destination with an address (as maintaining address of individual nodes is a large overhead). Instead of using a device based addressing scheme, data identification is used, and according to the data interest, results and data are exchanged.

Several data dissemination methods for wireless sensor networks are defined by protocols and most of these protocols are variants of the methods used in the internet or in the literature. The most referred data dissemination methods are as follows:

Classic Flooding: In classic flooding, a node wishing to disseminate a piece of data across the network starts by sending a copy of the data to all of its neighbors.

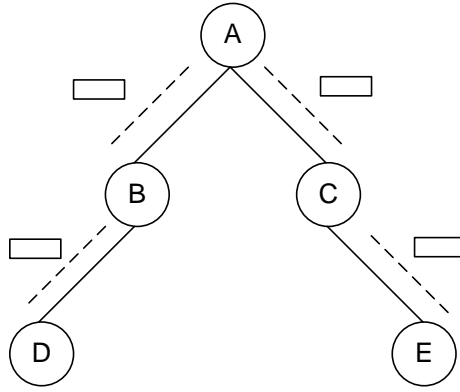


Figure 3.1 Classic Flooding

Whenever a node receives new data, it makes a copy of the data and sends the data to all of its neighbors, except the node from which it just received the data. In Figure 3.1, a sample communication scheme is given. A sends the message to its neighbors B and C. Then, B and C copy the message and send the message to their neighbors D and E respectively. The algorithm finishes when all the nodes in the network have received a copy of the message.

Gossiping: Gossiping [11], which uses randomization to conserve energy, is an alternative to the classic flooding approach. Instead of forwarding data to all its neighbors, a gossiping node only forwards data onto one randomly selected neighbor. If a gossiping node receives data from a neighbor, it can forward data back to that packet originating neighbor if randomly selects that neighbor.

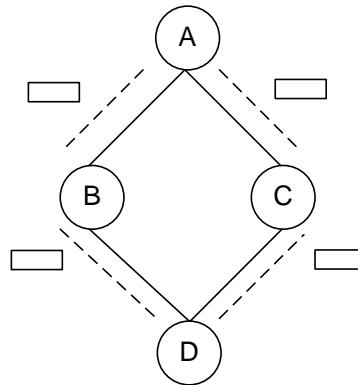


Figure 3.2 Packet Implosion

Whenever data travels to a node that is part of a high degree node density environment in a classic flooding network, more copies of the data start flowing around the network and implosion can occur. Implosion is the receiving of unnecessary duplicate packet traffic because of having same neighbor in transmission range. In Figure 3.2, D receives the same packet from both C and D

according to classic flooding because they transmit a copy of the message to their neighbor D. Gossiping avoids implosion by making one copy of each message at any node. While gossiping distributes information slowly, it dissipates energy at a slow rate as well.

Gossiping was originally introduced in discrete mathematics as a combinatorial problem in graph theory [12]. In gossiping, every person in the network knows a unique item of information and needs to communicate it to everyone else.

A popular formulation assumes there are n people, each one of whom knows a scandal which is not known to any of the others. They communicate by telephone, and whenever two people place a call, they pass on to each other as many scandals as they know.

Let $f(n)$ be the number of minimum calls necessary to complete gossiping among n people, where any pair of people may call each other. Then $f(1) = 0$, $f(2) = 1$, $f(3) = 3$ and according to [13], $f(n)$ is calculated as in Equation 3.1.

$$n \geq 4 \Rightarrow f(n) = 2n - 4 \quad (3.1)$$

In the case of one-way communication as in the case of sensor networks, the graph becomes a directed graph and the minimum number of calls becomes

$$n \geq 4 \Rightarrow f(n) = 2n - 2 \quad (3.2)$$

according to [14].

3.1.1 SPIN

Sensor Protocols for Information via Negotiation (SPIN) [15] is a family of negotiation-based information dissemination-based information protocols suitable for wireless sensor networks. SPIN focuses on the efficient dissemination of individual sensor observations to all the sensors in a network, treating all sensors as potential sink nodes. The design of SPIN grew out of the analysis of the strengths and limitations of conventional protocols such as classic flooding for disseminating data in a sensor network. Classic flooding is a very simple, straightforward protocol for information dissemination. However, it's inefficient in terms of energy. First, nodes send data to their neighbors regardless of whether or not they have already received

the data from another source. Second, sensor networks are expected to be very dense, multiple copies of the same data will be disseminated in the same region. Lastly, the protocol is not energy aware which would allow a sensor protocol to change its communication parameters depending on the energy left on the node.

To overcome these problems, SPIN nodes negotiate with each other before transmitting data. Negotiation helps ensure that only useful information will be transmitted. Meta-data is used to discriminate transmitted data from each other to avoid data replication in the network. It allows nodes to name the portion of the data that they are interested in obtaining. In order to add energy efficiency metric into SPIN, nodes poll their resources before data transmission. Each sensor node has its own resource manager that keeps track of resource consumption. This allows sensors to cut back on certain activities when energy is low.

SPIN family of protocols rests upon two basic ideas. First, to operate efficiently and to conserve energy instead of exchanging the data each node has, SPIN exchanges summary of the data each node has and the data they request. Second, nodes in a network monitor and adopt to changes in their own energy resources to extend the battery lifetime of the system.

Two versions of the protocol have been implemented to test SPIN. SPIN-1 includes a three way handshake protocol which uses advertise, request and data messages to exchange lost or new information among the nodes. Meta-data is used in request messages so that only nodes that have the requested information respond. SPIN-1's data exchange method is depicted in Figure 3.3. Node A advertises its data and node B responds with request message and then node A transmits data to node B. This process continues until all the nodes have the data.

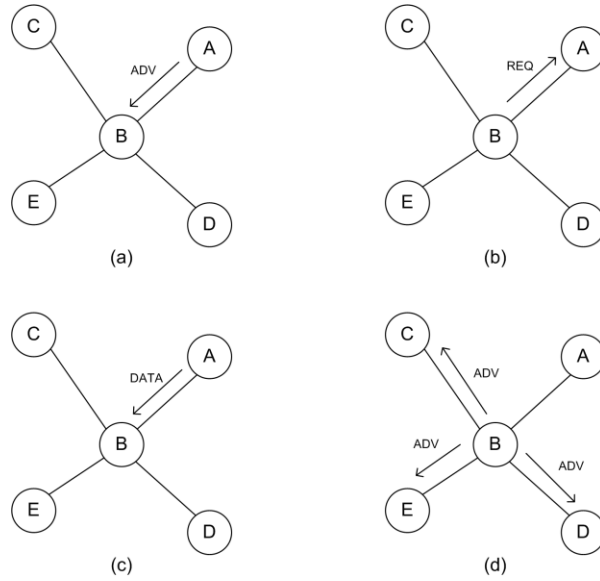


Figure 3.3 SPIN Data Exchange

SPIN-2 adds energy efficiency to SPIN-1. When a SPIN-2 node observes that its energy is approaching a low-level threshold, it adapts by reducing its participation in the protocol. In general, a node will only participate in a stage of the protocol if it believes that it can complete all the other stages of the protocol without going below the low-energy threshold. Similarly, if a node receives an advertisement, it does not send out a request if it does not have enough energy to transmit the request and receive the corresponding data.

SPIN proves that: Naming data using meta-data descriptors and negotiating data transmissions using meta-data successfully solve the implosion and overlapping region problems. SPIN protocols are simple protocols and they maintain no neighbor state which makes them suitable for mobile nodes.

3.1.2 Directed Diffusion

Directed Diffusion [16] is a data-centric paradigm for coordination of dense sensor nodes to collect data in a distributed fashion. Data gathered by sensor nodes is named by attribute-value pairs. A node requests data by sending interests for named data. Data matching the interest is then drawn down towards that node. Intermediate nodes can cache, or transform data, and may direct interests based on previously cached data. An important feature of directed diffusion is that interest and data propagation and aggregation are determined by localized interactions.

Type	: four-legged animal	// Detect animal location
Interval	: 20ms	// Send back events every 20ms
Duration	: 10 seconds	// For the next 10 seconds
Rect	: [-100,100,200,400]	// From sensors within range

Figure 3.4 A Sample interest description

Directed diffusion consists of several elements. Data is named using attribute-value pairs as in Figure 3.4. A sensing task is disseminated throughout the sensor network as interest for named data. The dissemination sets up gradients within the network designed to draw events. A gradient consists of a destination ID and data flow rate. Events start flowing towards the originators of interests in the form of attribute-value pairs along multiple paths. The sensor network reinforces one, or a small number of these paths as seen on Figure 3.5..

An interest is usually injected into the network at the node named sink in the network. A task for each named interest is created which is temporarily stored at sink node until the task's timer times outs. For each active task, the sink periodically broadcasts an interest message that contains the parameters listed in Figure 3.4 to each of its neighbors. Instead of transmitting at the original interval rate (20ms), sink node begins with a lower rate (1 sec.) to discover if any node can answer to the interest. A timestamp is appended to each interest message to track requests from the sink node so that messages with greater timestamps are preferred to older ones.

Every node maintains an interest cache in which each item corresponds to a distinct interest. Timestamp of the last received matching interest, sender ID, duration and several gradient fields per neighbor that contain data rate requested by specified neighbor are found in the cache.

After receiving an interest, a node may decide to re-send the interest to some subset of its neighbors. To its neighbors, this interest appears to originate from the sending node, although it might have come from a distant node. This is an example of local interaction. In this manner, interests diffuse throughout the network. Not all received interests are re-sent. A node may suppress a received interest if it recently re-sent a matching interest.

Interest cache is used for directing interests to correct locations. For example, if in response to an earlier interest, a node heard from some neighbor a data sent by some

sensor within the region specified by the interest, it can direct this interest to real destination using the its cache, rather than broadcasting to all neighbors.

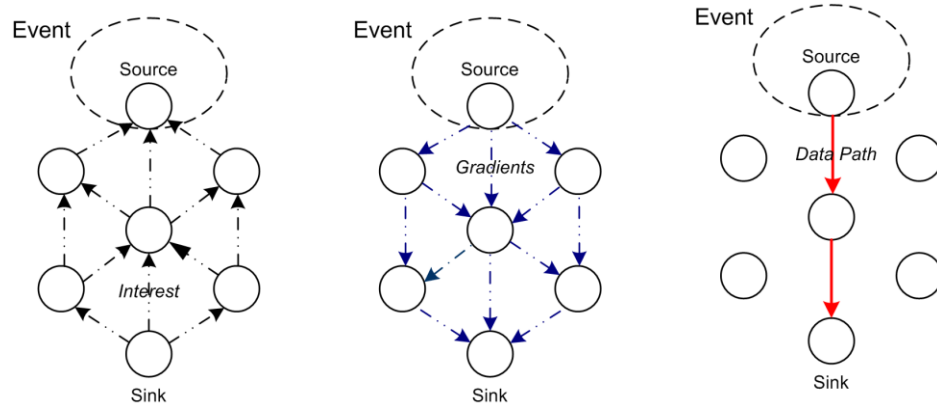


Figure 3.5 Interest dissemination

A scenario for interest dissemination is given in Figure 3.5. Sink node sends its interest in attribute-value pairs to its neighbors. Each neighbor records sending node to its cache and retransmits message to its own neighbors. When the interest reaches event location at source node, it computes the highest requested event rate among all its outgoing gradients and starts its sensing interface with the given event rate and transmits message to its neighbors at each interval. Sensor cache is used to prevent loops in the network. If a received data message has a matching data cache entry, the message is ignored. To resend a received data message, a node needs to examine the matching interest entry's gradient list. If all gradients have a data rate that is greater than or equal to the rate of incoming events, the node may simply send the received message to the appropriate neighbors. However, if some gradients have a lower data rate than others, then the node may downconvert to the appropriate gradient.

When an interest reaches the source node, it contains the interval timing of the sink node which is slower than the original interest. Once the sink node starts receiving message from the source node, it steadily increases the rate by retransmitting another interest message with a higher data rate. Each node that receives this announcement updates its entry and retransmits the interest. Nodes use probabilistic methods to choose among the neighbors that request higher data rates. This operation is called reinforcement. Using this algorithm, after some time the network will provide the sink node with a constant data rate data.

To sum up, in directed diffusion, all communication is neighbor-to-neighbor. Unlike traditional routers, each sensor node can interpret data and interest messages. Sensor

nodes do not need to have globally unique Ids as in internet protocols. Because every sensor can cache, aggregate and process messages, it is possible to perform coordinated sensing close to the sensed phenomenon. The aim of directed diffusion is to scale well to thousands of sensors. To achieve this local processing algorithms have been employed by trading some energy efficiency for increased robustness and scale

3.1.3 LEACH

Low Energy Adaptive Clustering Hierarchy (LEACH)[17] is a clustering based protocol that utilizes randomized rotation of local cluster base stations to evenly distribute the energy load among the sensors in the network. In LEACH, the nodes organize themselves into local clusters, with one node acting as local base station or station-head. LEACH includes randomized rotations of the high-energy cluster-head position such that it rotates among the various sensors in order not to drain the battery of a single sensor. In addition LEACH performs local data fusion to compress the amount of data being sent from the clusters to the base station.

Sensors elect themselves to be local cluster-heads at any given time with a certain probability. These cluster-head nodes broadcast their status to the other sensors in the network. Each sensor node determines to which cluster it wants to belong by choosing the cluster-head that requires the minimum communication energy. Once all the nodes are organized into clusters, each cluster-head creates a schedule for the nodes in its cluster. This allows the radio components of each non-cluster head node to be turned off at all times except during each node's transmit time, thus minimizing the energy dissipated in the individual sensors. Once the cluster-head has all the data from the nodes in its cluster, the cluster-head node aggregates the data and then transmits the compressed data to the base station.

However, being cluster-head drains the battery of that node. In order to spread this energy usage over multiple nodes, the cluster-head nodes are not fixed. The decision to become a cluster-head depends on the amount of energy left at the node. In this way, nodes with more energy remaining will perform the energy-intensive functions of the network. Each node makes its decision about whether to be a cluster-head independently of the other nodes in the network and thus no extra negotiation is required to determine the cluster heads.

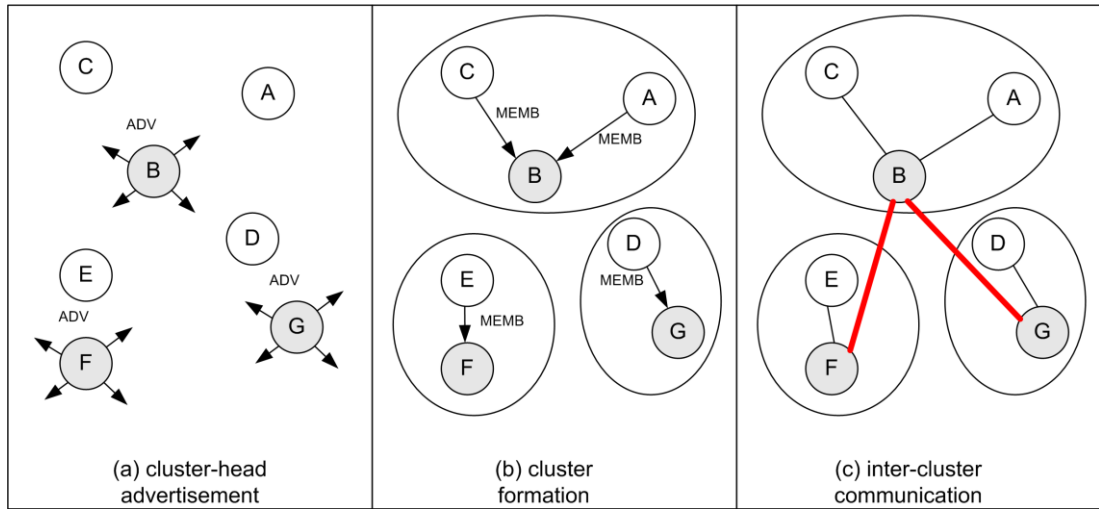


Figure 3.6 LEACH Cluster Formation

Cluster formation algorithm is given in Figure 3.6. Each node, that has elected itself as a cluster-head for the current round, broadcasts an advertisement message to the rest of the nodes in the cluster. The cluster-head uses CSMA-MAC protocol and all cluster heads transmit their advertisement using the same transmit energy. Each non-cluster head node decides the cluster which it will belong for this round. This decision is based on the received signal strength of the advertisement. After each node has decided to which cluster it belongs, each node transmits membership information back to cluster-head using a CSMA-MAC protocol. The cluster-head receives all the messages for nodes that would like to be included in the cluster. Based on the number of nodes in the cluster, the cluster-head node creates a TDMA schedule telling each node when it can transmit. This schedule is broadcast back to the nodes in the cluster. Nodes transmit their data during their schedule to the cluster-head. When all the data has been received, the cluster head node performs signal processing functions to compress data. Finally, processed data is transferred between cluster-nodes via a tree up to sink node.

From the simulations made, LEACH can achieve over a factor of 7 reduction in energy dissipation compared to direct communication with the base station, when using the optimal number of cluster-heads. The main energy savings of the LEACH protocol is due to combining lossy compression with data routing. In this case, some data from the individual signals is lost, but this results in a substantial reduction of the overall energy dissipation of the system.

3.2 Query Processing and Resolution

After the user requests arrive to sensor nodes via dissemination, these requests are first processed by the sensor node to see if the sensor node can match user requests. If a node has the capability for answering user request, the request is resolved by collecting sensing information and an answer to the query is generated and posted towards the user. This approach is one of the simplest ways of resolving and processing query. Sensor nodes usually take advantage of collaborative processing to resolve user requests so that smaller number of messages are transmitted in the network.

Mostly referred methods that are in use by many protocols are as follows:

Flooding based Query Resolution: The querier floods multiple copies of the query as in classic flooding mechanism for query dissemination and nodes with the relevant data then respond. Flooding can dominate the costs if the query is not continuous.

Expanding Ring Search (ERS) based Query Resolution: At stage 1, the querier will request information from all sensors exactly one hop away . If the query is not completely resolved in the first stage, querier will send a request to all sensors two hops away in the second stage. This process continues until query is resolved. When ERS is used, the energy saving for query resolutions that can be achieved within small number of hops is tremendous because of the less overhead of flooding of the environment. However, if the query can not be resolved within the near nodes, excessive flooding of the network will be the biggest issue for smaller hops.

3.2.1 TINYDB

TinyDB [18] is a query processing system for extracting information from a network of Tiny Micro threading Operating System (TinyOS) sensors. Unlike existing solutions for data processing in TinyOS, TinyDB does not require embedded C code for sensors to be written. Instead, TinyDB provides a simple, SQL-like interface to specify the data, along with additional parameters, like the rate at which data should be refreshed – much as in traditional databases. Given a query specifying data interests, TinyDB collects that data from motes in the environment, filters it, aggregates it together, and routes it out to a PC. TinyDB does this via power-efficient in-network processing algorithms.

The primary goal of TinyDB is to allow data-driven applications to be developed and deployed much more quickly than what is currently possible. Some of the features of TinyDB include:

Metadata Management: TinyDB provides a metadata catalog to describe the kinds of sensor readings that are available in the sensor network.

High Level Queries: TinyDB uses a declarative query language that lets the data to be described without requiring stating how to get it. This makes it easier to write applications, and helps guarantee that applications continue to run efficiently as the sensor network changes.

Network Topology: TinyDB manages the underlying radio network by tracking neighbors, maintaining routing tables, and ensuring that every mote in the network can efficiently and (relatively) reliably deliver its data to the user. A particular query for network topology is executed on motes, with results displayed in a special visualization.

Multiple Queries: TinyDB allows multiple queries to be run on the same set of motes at the same time. Queries can have different sample rates and access different sensor types, and TinyDB efficiently shares work between queries when possible.

Incremental Deployment via Query Sharing: TinyDB motes share queries with each other: when a mote hears a network message for a query that it is not yet running, it automatically asks the sender of that data for a copy of the query, and begins running it.

TinyDB system contains two applications: one application runs on the sensor platforms and another application runs on the PC side. A user requests his query using the Java application on the PC and this query is disseminated to sensor nodes and the application on the sensor platforms retrieve and return the requested information.

A sensor platform application consists of Sensor Catalog and Schema Manager, Query Processor, Memory Manager and Network Topology Manager. Sensor Catalog is responsible for tracking the set of attributes, or types of readings available on each sensor. The Query processor uses the catalog to fetch the values of local attributes, receives sensor readings from the neighboring nodes over the radio, combines and aggregates these values together, filters out undesired data, and outputs

values to parents. Memory manager is a heap memory manager implemented on top of static memory. TinyDB manages the connectivity of motes in the network, to efficiently route data and query sub-results through the network.

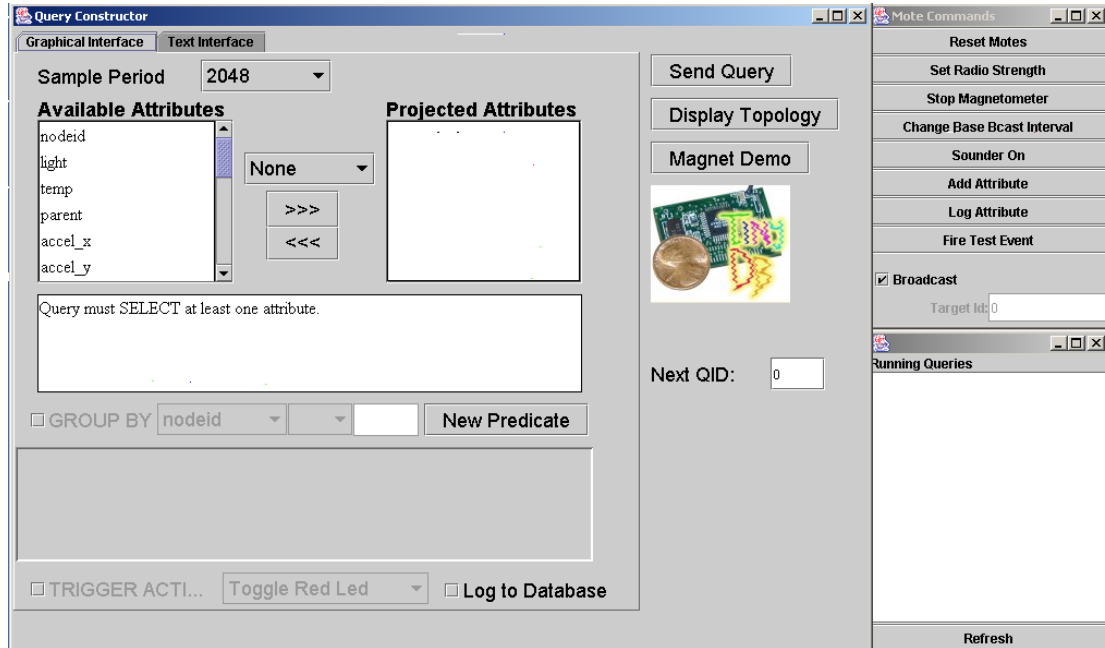


Figure 3.7 TinyDB GUI

TinyDB client application in Figure 3.7 running on the PC is used to construct and inject the queries and listen for results. A graph and table Graphical User Interface (GUI) is used to display individual sensor results.

TinyDB implicitly queries one single, indefinitely-long logical table called sensors. This table has one column for each attribute in the catalog, including sensor attributes, nodeIDs, and some additional introspective attributes that describe mote's states. This table conceptually contains one row for each reading generated by any mote. TinyDB's query language is based on SQL and named as TinySQL. TinySQL results are posed over rows generated by multiple sensors at one point in time.

SELECT	Temp
FROM	Sensors
WHERE	temp> threshold
TRIGGER ACTION	SetSnd(512)
EPOCH DURATION	512

Figure 3.8 Triggering SQL query

TinyDB also includes a facility for simple triggers, or queries that execute some command when a result is produced. A sample query is given in Figure 3.8 which

calls SetSnd function to give alarm when the temperature is over some threshold value and this value is checked every 512 seconds.

TinyDB includes the ability to run queries that log into the Flash memory of the motes. TinyDB provides commands for creating tables that reside in flash, for running queries that insert into these tables, for running queries that retrieve from these tables, and for deleting these tables. One query can log to a buffer at a time, and that new queries will overwrite data that was previously logged to a table. Currently, a query that selects from a Flash table and a query that writes to the same table can not be run. Logging of the query should be stopped using TinyDB Client utility prior to collecting data from flash table.

When running queries longer than 4 seconds by default, TinyDB enables power-management and time-synchronization. This means that each sensor is on for exactly the same four seconds of every sample period. Results from every sensor node for a particular query should arrive at the basestation within four seconds of each other. This time synchronization and power management enables long running deployments of sensors.

TinyDB aggregates results on the way to the sink node. TAG [19] is an aggregation service offered by TinyDB. It operates as follows: users pose aggregation queries from a powered, storage-rich base station. Operators that implement the query are distributed into the network by piggybacking on the existing ad hoc networking protocol. Sensors route data back towards the user through a routing tree rooted at the base station. As data flows up this tree, it is aggregated according to an aggregation function and value-based partitioning specified in the query.

In order for users to pose declarative queries, an SQL like programming language was designed.

SELECT	AVG(volume),room
FROM	Sensors
WHERE	floor = 6
GROUP BY	Room
HAVING	AVG(volume) > <i>threshold</i>
EPOCH DURATION	30s

Figure 3.9 A Sample TAG Query

A sample SQL query is given in Figure 3.9. In this query it's requested from sensor nodes to report the room number and microphone level of each sensor in floor 6 if

the average volume is greater than the threshold for 30s. It's quite similar to SQL queries but it has limited capabilities.

TAG implements three function: a merging function f , an initializer function i , an evaluator e , to implement aggregation services. In general f has the following structure:

$$\langle z \rangle = f(\langle x \rangle, \langle y \rangle) \quad (3.3)$$

$\langle x \rangle$ and $\langle y \rangle$ are multi-valued partial state records, computed over one or more sensor values, representing the intermediate state over those values that will be required to compute an aggregate. $\langle z \rangle$ is the partial state record resulting from the application of function f to $\langle x \rangle$ and $\langle y \rangle$. For example, if f is the merging function for AVERAGE, each partial state record will consist of a pair of values: SUM and COUNT. f is specified as follows, given two state records $\langle S_1, C_1 \rangle$ and $\langle S_2, C_2 \rangle$:

$$f(\langle S_1, C_1 \rangle, \langle S_2, C_2 \rangle) = \langle S_1 + S_2, C_1 + C_2 \rangle \quad (3.4)$$

The initialized i is needed specify how to instantiate a state record for a single sensor value; for an AVERAGE over a sensor value of x , the initializer $i(x)$ returns the tuple $\langle x, 1 \rangle$. Finally, the evaluator e takes a partial state record and computes the actual value of the aggregate. For AVERAGE, the evaluator $e(\langle S, C \rangle)$ simply returns S/C .

Aggregates are classified according to four categories according to their state requirements, tolerance of loss, duplicate sensitivity and monotonicity:

Duplicate Insensitive/Sensitive Aggregates: Duplicate insensitive aggregates are unaffected by duplicate readings. Duplicate sensitive aggregates will change when a duplicate reading is reported.

Exemplary/Summary Aggregates: Exemplary aggregates return one or more representative values from the set of all values. Summary aggregates compute some property over all values.

Monotonic Aggregates: These aggregates have the property that when a function f is applied to two partial state records for all resulting values, it will be greater or lower than each of the evaluation of pair's values. This proves increasing or decreasing values for aggregate results.

Distributive/Algebraic/Holistic/Unique/Context-sensitive Aggregates: Depending on the function a pair of values has to be carried. For example AVERAGE function requires the number of elements used to compute the result and the result to further continue in processing. Distributive aggregates don't require other data to calculate result; therefore, size of the partial records is the same as size of the final record. COUNT, MAX and MIN are an example of distributive aggregates. Algebraic aggregates require intermediary state information to continue operation. AVERAGE function is an example of algebraic aggregate. Holistic aggregates require whole values to be kept together prior to computing the result. MEAN is one of these operators. Unique aggregates are similar to holistic aggregates, except that the amount of state that must be propagated is proportional to the number of distinct values in the partition. In Context-sensitive aggregates, the partial state records are proportional in size to some property of the data values in the partition. Many approximate aggregates are content-sensitive. Fixed-width histograms and wavelets are examples of these operators.

Queries in TAG contain named attributes. When a TAG sensor received a query, it converts named fields into local catalog identifiers. Nodes lacking attributes specified in the query simply tag missing entry as NULL. This increases the scalability as not all the nodes are required to know global knowledge of all attributes. Attributes can be sensor values, remaining energy or network neighborhood information.

TAG computes aggregate in network whenever possible to decrease the number of message transmissions, latency and power consumption. Given the goal of decreasing the number of transmitted messages, during the collection phase each parent waits for some time period prior to transmitting its own message in order to aggregate with the children nodes' responses. How long each node waits for other nodes' responses is $(\text{EPOCH DURATION})/d$, where d is the maximum depth of the tree.

In order to group received data group id is tagged to each sensor partial state record. So that, response data is aggregated for the nodes with same group id. When a node receives an aggregate from a child, it checks the group id. If the child is in the same epoch as the node, it combines the two values. If it's in another epoch, it stores the value of the child's group along with its own value for forwarding in the next epoch.

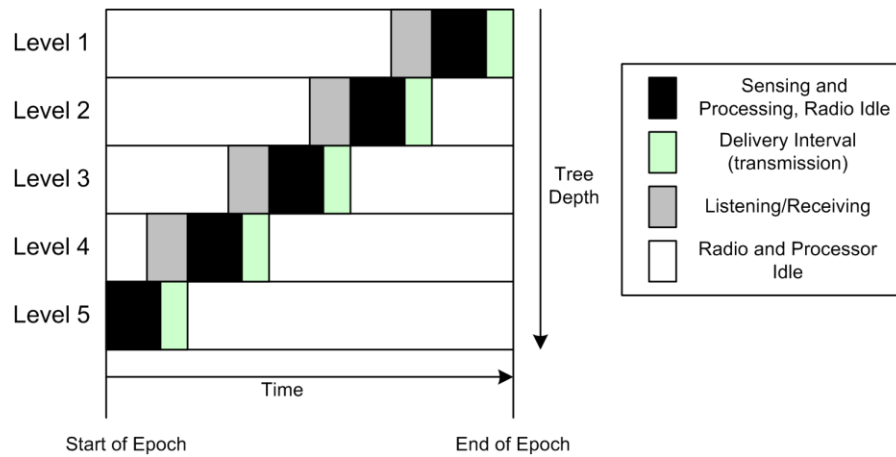


Figure 3.10 Partitioning of time into EPOCHS

For the query in Figure 3.9, computation is applied for each sensor at each room and results are transmitted to upper nodes. Base station combines each room's information to display to the user.

The principal advantage of TAG is that it dramatically decreases the number of messages by aggregation and employing distributed data processing. It can also tolerate disconnections since it carries partial state information. Any node can join query processing from the middle of the operation. By explicitly dividing time into epochs as in Figure 3.10 while waiting for other nodes' responses to arrive, the CPU can be set to idle period during this time. But, the depth of the tree has to be known for this principle to be made successful. However, waking up the processor will require synchronization to be employed.

Finally, real users of sensor networks are most likely not sophisticated software developers. Therefore, TinyDB has been supported by various toolkits [20] for easy access of data. The complexity of sensor network application development must be reduced and deployment must be made easy to ensure the success of sensor network technology in the real world.

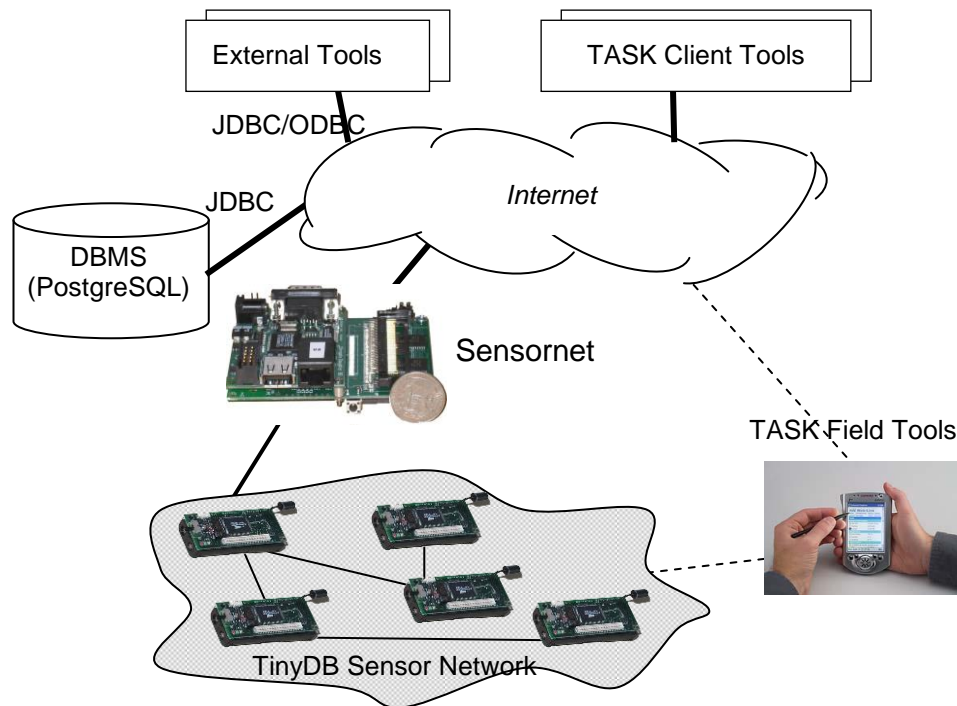


Figure 3.11 The TASK Architecture

The architecture of Tiny Application Sensor Kit (TASK) is depicted in Figure 3.11. TASK consists of the following components:

TinyDB based sensor network: allows traditional programs to interact the sensor network through a declarative SQL-like interface

TASK Server: a server process running on a sensor network gateway that acts as a proxy for the sensor network on the internet.

TASK DBMS: a relational database that stores sensor readings, sensor network health statistics, sensor locations and calibration coefficients, etc.

TASK Deployment Tool: helps users record sensor node metadata.

TASK Configuration Tool: helps users choose data collection intervals and data filtering and aggregation criteria.

TASK Visualization Tool: helps users monitor the network health and sensor readings.

TASK Field Tool: running on a PDA help users diagnose and resolve problems in certain areas of the network in the field.

3.2.2 COUGAR

Cougar [21] is a query layer for sensor networks. The query layer accepts queries in a declarative language that are then optimized to generate efficient query execution plans with in-network processing which can significantly reduce resource requirements.

Cougar is motivated by three design goals. First, declarative queries are especially suitable for sensor networks. Clients issue queries without knowing how the results are generated, processed and returned to the client. Second, it is very important to preserve limited resources such as energy and bandwidth. Since sensor nodes have the ability to perform local computation, part of the computation can be moved from the clients and pushed into the sensor network, aggregating records or eliminating irrelevant records. Third, different applications usually have different requirements, from accuracy, energy consumption to delay. For example, a sensor network deployed in a battlefield or rescue region may only have a short life time but a higher degree of dynamics. On the other hand, for a long term scientific research project that monitors an environment, power-efficient execution of long-running queries might be the main concern. More expensive query processing techniques may shorten processing time and improve result accuracy, but might use a lot of power. The query layer can generate query plans with different tradeoffs for different users.

The component of the system that is located at each node is called query proxy. Architecturally the query proxy lies between the network layer and the application layer and the query proxy provides higher level services through queries. Gateway nodes are connected to components outside of the sensor network through long-range communication and all communication with users of the sensor network goes through the gateway node.

SELECT	{attributes, aggregates}
FROM	{SensorData S}
WHERE	{Predicate}
GROUP BY	{attributes}
HAVING	{predicate}
DURATION	time interval
EVERY	time span e

Figure 3.12 Query Template

Declarative queries are the preferred way of interacting with a sensor network. The queries having the form in Figure 3.12 are considered. It is very similar to the SQL

language and it has limitations when compared to SQL. One difference between the query template and SQL is that query template has additional support for long running, periodic queries. The DURATION clause specifies the life time of a query and the EVERY clause determines the rate of query answers.

A simple aggregate query is an aggregate query without GROUP BY and HAVING clauses. In order to compute these aggregates several processing such as in-network aggregation has to be done. In order to process data in-network, several sensor nodes transmit the packet to a central node named leader-node which calculates the aggregates of incoming messages. There are three approaches that can be taken to collect sensor data at leader node. Messages can be directly delivered to leader node using ad hoc routing protocol, messages can be merged into same packet to limit amount of packet transmitted and partial aggregation on the way to the central node can be done by intermediary nodes.

Synchronization is needed if the messages are to be merged and some duplicate sensitive operators such as SUM and AVERAGE require data to be transmitted once. Synchronization is used to determine for each node in each round of the query to determine how many sensor readings to wait for and when to perform packet merging or partial aggregation.

Since query processing facility has been designed as a layer, COUGAR assumes that several ad hoc routing protocols with modifications can be used for delivery of the messages. Ad hoc On Demand Distance Vector Routing Protocol (AODV) [22] protocol has been used with extensions for simulations. According to COUGAR approach, routes are set up at initialization phase and each message carries the hop count of the message. Each node records the message receive ID as a parent node and a reverse path to the leader is set up. Two methods are used to maintain the tree. Local repair is used when a broken link is detected. Depth of the tree with sequence number is used between nodes that are spatially close to find a new parent in the case of a communication failure. Another method is to reconstruct the tree whenever the number of messages expected reach below some user defined threshold.

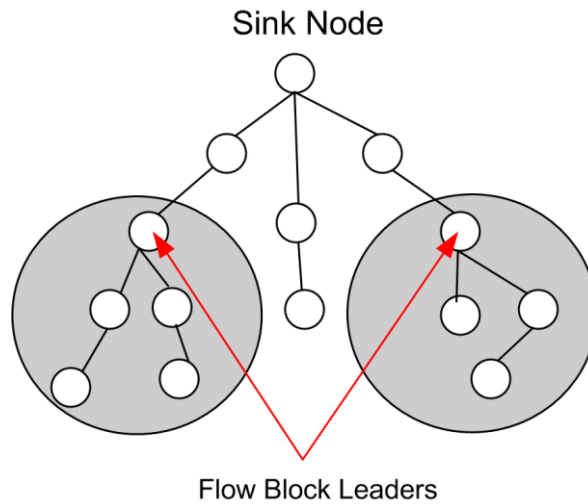


Figure 3.13 A Cougar Query Plan

In order to resolve queries like “What is the minimum average temperature during the next seven days?” two levels of aggregation has to be done. First the average temperature has to be computed and then the minimum operator has to be applied. In order to resolve these kinds of queries query plans are used. A query plan is needed to compute complex aggregate queries that a user poses. A query plan decides how much computation is pushed into the network and it specifies the role of and responsibility of each sensor node, how to execute the query, and how to coordinate the relevant sensors. A query plan is constructed by flow blocks, where each flow block consists of a coordinated collection of data from a set of sensor nodes at the leader of the flow block as depicted in Figure 3.13. The task of a flow block is to collect data from relevant sensor nodes and to perform some computation at the destination or sensor internal nodes. A flow block is specified by different parameters such as the set of source sensor nodes, a leader selection policy, the routing structure and the computation that the block should perform. Each flow block is called a cluster and maintained by some heartbeat messages transmitted by the leader of the flow.

Several optimizations can be applied to query plan construction such as creating flow blocks that are sharable between different queries and use of join operator which enforces two conditions coming from different data flows to be true before returning a value. Join operator represents a wide range of possible data reductions. Depending on the selectivity of the join, it is possible to reduce the resulting data size.

3.2.3 ACQUIRE

Active Query Forwarding in Sensor Networks (ACQUIRE) [23] is a mechanism for obtaining information in sensor networks. In ACQUIRE, an active query is forwarded throughout the network, and intermediate nodes use cached local information in order to partially resolve the query. When the query is fully resolved, a completed response is sent directly back to the querying node.

With large scale networks of energy-constrained sensors it is not feasible to collect all measurements from each device for centralized processing. Due to energy constraints it is desirable for much of the data processing to be done in-network, and this has led to the concept of data-centric information routing, in which queries and responses are for named data. Depending on the applications, there are likely to be different kinds of queries. These queries can be categorized as follows:

Continuous queries: result in extended data flows.

Aggregate queries: aggregation of information from several sensors is done.

Complex queries: consists of several nested or batched subqueries

Queries for replicated data: response to a given query can be provided by many nodes.

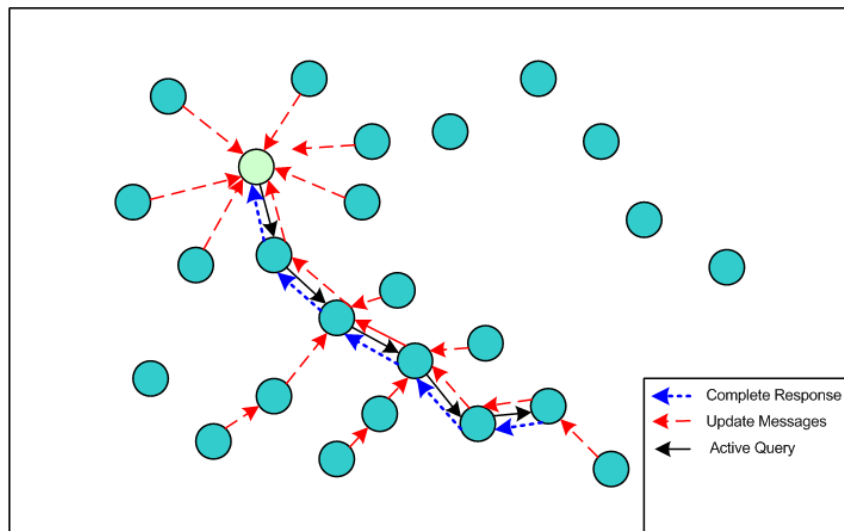


Figure 3.14 ACQUIRE Query Resolution

The principle behind ACQUIRE is to inject an active query packet into the network that follows a random or guided trajectory throughout the network. At each step, the node which receives the active query performs a triggered, on-demand, updated obtaining information from all neighbors within a look-ahead of d hops as in

Figure 3.14. New updates are triggered reactively by the active node upon reception of the active query if the current information it has is obsolete. As this query progresses through the network it gets progressively resolved into smaller and smaller components until it is completely solved and is returned back to the querying node as a completed response through the return path or shortest path. The choice of a next hop to forward the message is done randomly or by intelligence based on other information that guarantee further resolution of the query.

ACQUIRE is likely to perform in an energy-efficient manner compared to other approaches on complex, one-shot and non-aggregate queries for replicated data. In particular, ACQUIRE with optimal parameters performs many orders of magnitude better than flooding-based schemes. %60 Energy savings can be achieved when compared to ERS.

4 SeMA QUERYING PROTOCOL FOR MICRO-SENSORS (SQS)

SeMA, which is an acronym for A Session Based Mobile Ad Hoc Network Architecture, is a project supported by TUBITAK under grant no: 101E037/ EEAG-AY-41. It has started as an ad hoc network routing evaluation work [24]. The project then evolved into an architectural proposal that tries to address some important problems in the mobile ad hoc networking environment. Under the assumption of an existing wireless data link layer protocol, the architecture covers a clear definition of an ad hoc network as a session, and defines agents of session, routing and service in order to maintain an ad hoc network to discover a service and use it. For the purpose, an expressive definition of services is considered and SeMA aware application scenarios are being developed.

In this study, SeMA architecture has been extended to the sensor networks with design and implementation on a real time operating system for sensor nodes so that sensing capabilities of the nodes are expressed to users as services of the mobile ad hoc network. For example, a user which has access to a sensor node that is part of the sensor network in the forest, can announce temperature reading capability of the mobile node as a service to other users of the mobile ad hoc backbone and any user in the ad hoc backbone can bind to this service to fetch the results.

SeMA architecture defines services as attribute-value pairs as recommended in other studies [23], [21]. To map this capability to sensor networks a general purpose query processing and dissemination protocol is defined in which capabilities of underlying network such as immediate or aggregated delivery and reliable event delivery is expressed to users as customizable set of options so that a user's request can be resolved by an appropriate combination of these options.

This protocol named as SQS (SeMA Scalable Querying Protocol for Micro-Sensors) hereafter is the lowest tier of the architecture and is completely independent of other protocols in the architecture. It's currently assumed that specialized nodes which serve as a gateway between the tiers of the architecture will provide the necessary conversion between the protocols.

4.1 SeMA Architecture

Dynamic service discovery and late binding to the most appropriate network service will be the key characteristics of future ad hoc network. Mobile nodes dynamically establish routes among themselves to form a network “on the fly”. SeMA [25] incorporates a service discovery protocol based on a node mobility and service definition model formally specified in eXtensible Markup Language (XML) [26]. The protocol employs a session-based approach to service access whereas the discovery algorithm returns both the service location and the route information for late binding.

SeMA is an enhanced client-server architecture in which clients are supported to adapt themselves to the new network resources they discover “on the fly”. The nodes get service information either by catching the periodically advertised services or by generating a service request. Applications establish light-weight sessions with the resources using the address returned to them. Binding is done when the service is needed.

SeMA is a cross-layer protocol, capable of operating on generic wireless data link layer protocols, such as IEEE 802.11. SeMA addresses issues like service announcement, binding, session management and data routing. The monitoring of sensor networks is an application of SeMA architecture. The terrain of the sensor deployment area is assumed to be suitable for navigating by means of some mobile units, which form the information retrieval backbone of the overall monitoring application [27]. These mobile units are either wireless equipment carrying livings or autonomous robots as seen in multi robot exploration studies. The architectural view of the system is shown in Figure 4.1.

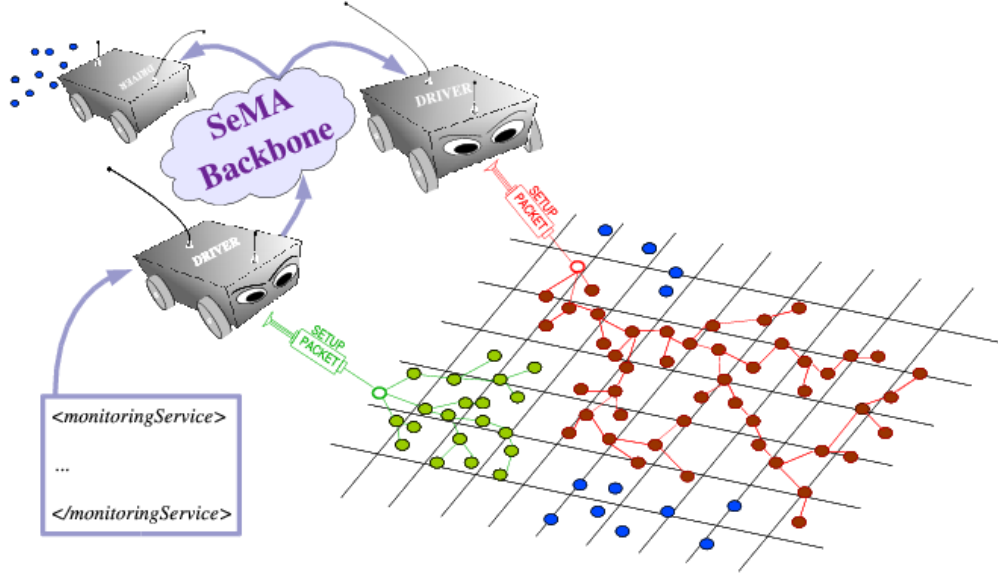


Figure 4.1 SeMA Architecture

An instructive example sensor network aims to detect the amount of hazardous chemicals that have spread over an urban environment, because of a leak during transportation or explosion of an on-site reactor. Wireless communication capable, chemical material density measuring sensors are deployed over the polluted area in an ad hoc fashion. Hazardous material density is the objective to be monitored from a safe distance. For the information retrieval, autonomous robots are released to the area, where each will initiate temporal sensor tree topologies to disseminate and gather results of a query created by the monitoring application. The monitoring application running host and the autonomous robots form a wireless backbone network, with longer transmission and reception capable radios and better energy capacity holding batteries compared to sensor node's batteries. Robots of the backbone network relay each other to announce the queries of monitoring service or to collect results of the queries back to monitoring application.

All available resources on the SeMA network are considered as services. They play a major role on SeMA network, since mechanisms of the protocol are built with the aim of providing means to access those services. Specification of a service should include necessary information for SeMA clients to determine whether the service fulfills clients need. Thus, services are modeled with a generic name, and following attribute-value pairs. This definition is sufficient for a host to discover a required service without any bindings beforehand. Details of XML processing of services and ad hoc routing in the backbone can be found in [27].

In sensor network application, clients of the ad hoc network backbone become middleman between monitoring application which is defined in XML in Figure 4.2. They participate in forming query driven sensor trees adaptively and backbone features are used to announce queries and return collected results.

```
<service name = "monitoring application">
  <keyword attribute ="validUntil">20031128193044EEST</keyword>
  <keyword attribute = "queryPredicate">sensor:read[value>100 and order(value)<5]</keyword>
  <keyword attribute = "resultFunction">fn:concat</keyword>
</service>
```

Figure 4.2 XML definition of a monitoring service

The components of sensor network are as follows:

Client: Clients are mobile nodes modeled with two wireless network interfaces; communication interface, which runs SeMA ad hoc backbone protocol, is used to fetch monitoring service instances for query attributes. All data required for driving a sensor network topology such as the query, query time boundaries, runtime characteristics, geographic region boundaries etc. are encapsulated in the service attributes. Upon fetching a service, mobile client pauses its movement, activates the sensor *driver* process and goes to sleep. The driver process initiates a rooted tree network among a subset of all sensors in the region via the node's second wireless network interface which is a short range radio capable of communicating with sensor node radios in the same transmission range. This interface is used to disseminate the query in a controlled way. During *topology construction* and *query running* phases, the driver node is assumed to be fixed; not moving. Upon collecting event data from the sensor network, the driver wakes up and establishes an implicit SeMA session with the monitoring service to transmit collected query responses.

Driver: Clients bind to a monitoring service with an implicit intention to get query and network parameters in service definition. Further, sensor driver process on the client is activated with these parameters to convert XML compliant query parameters into a bitwise coded format for dissemination through the sensors.

Workers: Sensors of the embedded network are called workers.

4.2 Query Resolution

Queries for data of interest are transmitted through the backbone in service announcement packets. Service definitions contain an XQuery [28] predicate, a result

function and Query Timeout Period (QTP), as well as geographic region boundaries of the area of interest. XQuery predicate is used to extract the readings that interest the monitoring application. Then, if specified in the service, these readings may be processed via the given resultFunction. The actual query result to be submitted is the returned data from this XQuery function. XQuery specifies more than 200 functions (including functions in SQL) that include numerical processing, data aggregation (Sum, Average (avg) , Minimum (min) , Maximum (max)), string operations (string-join, starts-with, ends-with), pattern matching (matches, replace, tokenize) etc. and more. By using XQuery in value fetching and processing, sensor querying process conforms to XML standard, from monitoring application down to the sensor nodes.

Temporarily posed sensor drivers convert service parameters and XQuery predicate to a bitwise coded format. Coded query is sent to sensors in the payload field of setup packet. In this model, sensor nodes are assumed to be very simple equipments with limited processing and battery power. Setup packet initiates a tree topology network in the area of interest among the sensors those have accepted the query.

XML service attributes are defined as:

ValidUntil: Query validity time.

QueryPredicate: An XQuery predicate that will filter out the desired sequence of values from the sensor readings. To be more specific, the given predicate instance will fetch the last five readings that carry numerical values greater than 100.

ResultFunction: A built-in or custom XQuery function that will be applied to the resulting sequence of the given predicate. What the function returns is to be sent to the client sensor node driving the tree. For the example instance, the sequences that are found by the predicate will be concatenated to form a compact resulting string of values.

A sample XML instance is given in Figure 4.2.

4.3 Query Definition

After being scattered around the terrain, sensor nodes operate in low power radio listening mode to receive messages from the driver. Queries are used to request data from sensor nodes. They can be used to collect data about the status of nodes or querying the environment via sensing interfaces. Therefore, queries that can be run

on sensor nodes are categorized as Sensing Related} and Node Related. When a sensing related query is made, it is indicated that the monitoring application is rather interested in data readings obtained from sensing interfaces than node status.

Example: Retrieve the daily average temperature for the next seven days.

Node related queries are used to set or get the constant values found on the nodes. Since sensor nodes are typically state machine oriented nodes, node related queries are used to externally force a state transition or a parameter calibration on the node. When a node related query is made, it is indicated that the application is rather interested in monitoring the network for topology maintenance than sensor interfaces.

Example: set the transmission power to %65 duty cycle

4.4 Compilation Functions

Raw data readings retrieved from sensors for some duration may not give enough idea about monitored environment. Compilation Functions applied to raw readings reduce the number of responses processed by the driver. Example: Alert the fire brigade when the temperature readings in the forest are over 70 degrees.

Compilation functions are used with Sample Count (SC) and Sampling Period (S) parameters to obtain more accurate query data on the node. SC defines the numbers of sensor reading samples to be taken by the node and S defines the time interval between samples. Nodes generate one response packet for every SC samples. Each response packet contains a compiled data of these samples as defined in the query predicates. This results in fewer response packets to be returned to the driver and consequently results in less power consumption for packet transmission overall.

4.5 Query Types

Query Processing Algorithm handles a number of different query types. Query specifies how data should be retrieved from a node and when a response packet should be generated. Three attribute classes are defined:

- **Response attempt:** is either continuous or one shot
- **Response generation method:** is either simple or complex

- **Response transmission:** is either aggregated or non-aggregated.

Descriptions of query attribute types are given below:

Continuous Queries: last for query timeout period and may result in multiple response packets from sensor nodes. These queries are used for periodic data collection. Example: Report daily temperature for a week at most three hops away.

One Shot Queries: result in a single response packet from every sensor node. These queries are used to receive one compiled result from sensors. Example: Report temperature readings from all nodes.

Simple Queries: return raw data readings from sensors without applying any compilation function. Example: What is the battery level? Or is sensing interface on?

Complex Queries: contain a compilation function such as sum, total, min, max of samples, to be applied to a number of raw sample readings. Example: Return temperature reading if after taking 3 samples at 3 min intervals any value is greater than 100 degrees.

Aggregate Queries: delay packet transmission at intermediate nodes during response delivery. Nodes keep responses from their children until their send buffer is full or timer expires. One network packet containing all responses in the buffer is transmitted. Our architecture currently supports max of four responses in a single data packet. A typical query example which may benefit from this type of aggregation: What is the number of nodes in the area?

Non-Aggregate Queries: return each response immediately. As soon as a packet is generated, or received by the node, it is relayed towards the root. Example: Return immediate response if at least one reading of 5 samples taken at 1 second intervals is over 70 degrees.

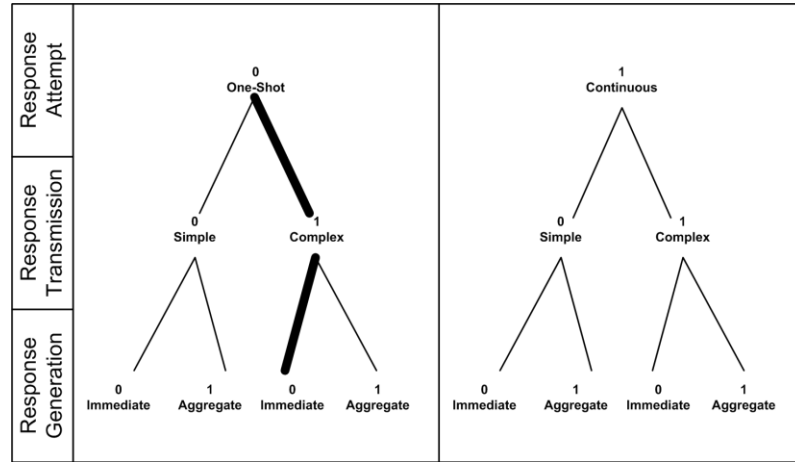


Figure 4.3 Query Driven Network Types

Sensor networks are classified with three binary digits encoding the query attributes of the network. For example: **010** type network processes a one shot, complex and non aggregate query. This network generates zero or one response from a simple reading on each connected sensor, resulting at most **N-1** network packet to be forwarded towards the root where N is the number of nodes. Additionally, each data packet is relayed separately via intermediate nodes. Figure 4.3 illustrates query driven network types.

4.6 Query Distribution

A Sensor network is created by the distribution of a setup packet as in Figure 4.4 among sensors and destroyed after the query is timed out or an abort packet is received. RF is a broadcast medium and each transmitted setup message is received by all neighboring nodes and disseminated with modifications at each hop. How far this broadcast message will be forwarded is given by the Flooding Degree parameter. Flooding is stopped when the Hop Count (hc) parameter becomes equal to flooding degree. Each node that receives setup packet decides to participate in the query network or not according to its battery level. Sensor nodes are allowed to join one network at time and each query network is uniquely identified by its Region-ID and Driver-ID pair.

Sensor node generates a local ID for itself upon receiving setup packet. Acknowledge flag in the packet is used to specify event delivery mechanism and concatenate flag is used to force piggybacking.

4.7 Query Processing

All participating sensors either act as intermediary routers that forward others response messages or both a query resolving and a message forwarding node. Nodes receiving setup packet inspect the requested query parameters and initiate a query resolving process if they have the requested capabilities. For sensing related sub queries, relevant sensing interface is switched on and for node related sub queries; required state data collection operation is activated.

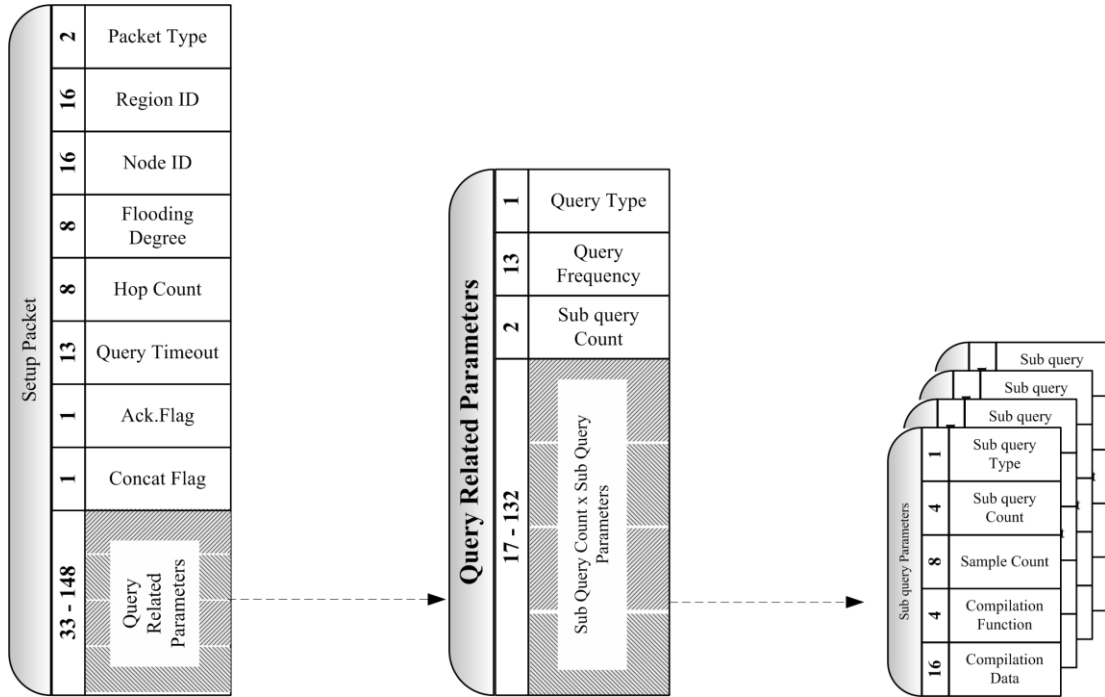


Figure 4.4 Setup Packet

As it can be seen from Figure 4.4, setup packet is a variable length packet that consists of one global query related parameters and several sub query parameters.

Query parameters are sent to sensors in the query related parameters part of the setup packet; *acknowledge flag-ack* is used to specify reliability requirement and *concatenate flag-concat* is used to force data buffering for message concatenation. Some applications might require individual results whenever they are ready, causing more traffic in the network and worse battery consumption whereas others might be interested in overall results. With *concatenate flag* set, the node collects responses from its children until the send buffer is full or the timer expires and sends the full buffer in a single message. *Sampling Period (S)* as explained before states how often data should be collected from sensing interface, *Query Type* specifies whether this

query is a continuous query or one-shot query. *Sub query Count* states the number of sub queries encapsulated into the setup packet.

Since data gathering interval requirements for sub queries can be different, time related information is encoded as Time Unit structure given in Figure 4.5.

3 bit	10 bit
Time Type	Time Value

Figure 4.5 Time Unit Structure

Time unit is represented in 3 bits which allow time type to be set to from microsecond to year scale. 10 bit value allows time value to be set up to 1024. So that very large scale timing values can be coded with 13 bits. All time related parameters of the protocols are represented in the same time unit structure in Figure 4.5. The coding of time types is also given in Table 4.1.

Table 4.1 Time unit encoding

Time Type	Value
Microsecond	0
Millisecond	1
Second	2
Minute	3
Hour	4
Day	5
Month	6
Year	7

Sub queries are also categorized into two types; sensing related and node related. *Subquery type* tells whether this query is sensing unit related or node related. Sensing related sub queries collect data from sensing interfaces and apply compilation function if requested. Node related queries allow the state of the node to be changed prior to sensing or retrieving node state as a part of the query.

Sub query Context field is a joint field. For sensing related sub queries, sub query context field is used to specify for which sensing interface this query is intended to and for node related sub queries, sub query context field is used to tell the function that will get or set the values. Encoding of sub query context field is shown in Figure 4.2.

Table 4.2 Sub query Context Encoding

Sub query Context	Sensing Related Value	Node Related Value
0000	Photo Sensor	Node ID
0001	Voltage Sensor	Parent ID
0010	Temperature Sensor	Neighbor ID
0011	Accelerometer Sensor	Battery Level
0100	Humidity Sensor	Packets Sent Per Node
0101	Microphone Sensor	Retransmission Count
0110	Magnetometer Sensor	Sensor Type
0111	Pressure Sensor	N/A

Compilation Data field can be redundant for some of the compilation functions and might be missing in the setup packet. This field is also used with node related queries in a jointly manner to specify the value to be set when we want to change a constant on the node.

For sensing related queries single raw data reading may not give enough idea about the environment. A number of raw samples may be needed to compile data. SQS query parameters specify how many raw samples must be taken (*sample count-SC*) and how frequent these readings have to be obtained from the sensing interface at each sampling period.

Sixteen compilation functions are defined for local query optimization. These functions are used with SC and S parameters to obtain more accurate query data on the node. Compilation functions and their encodings are given in Table 4.3.

Table 4.3 Compilation Function Encodings

Compilation Function	Value
Sum	0001
Count	0010
Average	0011
Min	0100
Max	0101
Greater	0110
Lower	0111

Nodes generate one response message for every SC samples. Each response packet contains a compiled data of these samples as defined in the query predicates. This generates fewer response packets to be returned to the driver, resulting in less power consumption for packet transmission overall.

Table 4.4 Sample Queries

Query Type	Sample Query	Network Parameters	
001	Return Node ID and Battery Level	QTP S SC	= x = x = x
010	Return Temperature reading if after taking 3 min sampling period any sample value is greater than the given threshold	QTP S SC	= 9 minutes = 3 = 3 minutes
100	Return Temperature reading for the next 7 days after taking 10 samples at 6 min sampling period	QTP S SC RGP QCycle	= (7*24*60)min = 6 minutes = 10 = 60 minutes = (7x24x60)/60=164
110	For the next 3 months return temperature reading if the average of 5 samples taken at 1 hour sampling period is greater than the threshold value.	QTP S SC RGP QCycle	= (90*24) hours = 1 hour = 5 = 5 hours = (90x24)/5=432

Compilation functions are the operators applied to the raw sensor readings. Threshold value is to derive results. Compilation functions are also assumed to be statically built on nodes and driver nodes. Compilation function "0000" has a special meaning and is used to specify whether the query is simple or not.

Table 4.4 gives some sample queries and the corresponding network parameters extracted from query attributes.

4.8 Response Generation

When a sensor node receives query parameters with a setup packet, it sets up a one shot query timeout timer to be fired at QTP and a continuous sampling period timer to be fired at each S time. The sampling period timer algorithm is given in Figure 4.6.

With each sampling period timer tick, if the sub-query is sensing related relevant sensing unit is switched on, the result is gathered and SC of the sub-query is decremented by one. This process continues until QTP is reached or SC of all sub queries reach zero which indicates that a response packet should be generated. If Continuous flag is not set, sampling period timer is stopped after packet transmission in order to save energy by avoiding unused timer interrupts.

```

SamplingPeriodTimer.fired()
begin
  if (!SubqueriesCompleted())
  begin
    forall subqueries begin
      if SubqueryCompleted()
        DisableSensingInterface()
      end
      if (ThereAreIncompleteSubqueries())
        // Request data from sensing interfaces
        StartCollectingDataFromSensingInterfaces()
      end
    end
  else begin
    PrepareAndPostResponseMessage()
    if (!ContinuousQuery)
      StopFrequencyTimer()
    end
  end
end

```

Figure 4.6 Sampling Period Timer Algorithm

As stated before, in active state, query nodes either generate query data or relay others data. In either case, nodes propagate messages towards upper levels of the tree by transmitting packets to their parents. They get parent ID and Region ID during setup packet processing. Nodes process messages only if their local region ID and node ID is equal to the region ID and message destination ID in the received packet. This allows simultaneous queries to be run in the same physical region without query networks interfering each other's processing.

Since proposed architecture supports up to four simultaneous sub queries, sub query responses have to be synchronized with each other on the sensor node. The network setup protocol supports different sample counts to be set up for different sub queries. As mentioned before, nodes wait for all sub queries to be resolved in order to generate a response message. The protocol sets a single Response Generation Period (RGP) for the whole network. With every sampling period timer tick, it is checked if all sub query sample counts has reached zero or not. Completed sub queries switch off their sensing interface and go to sleep.

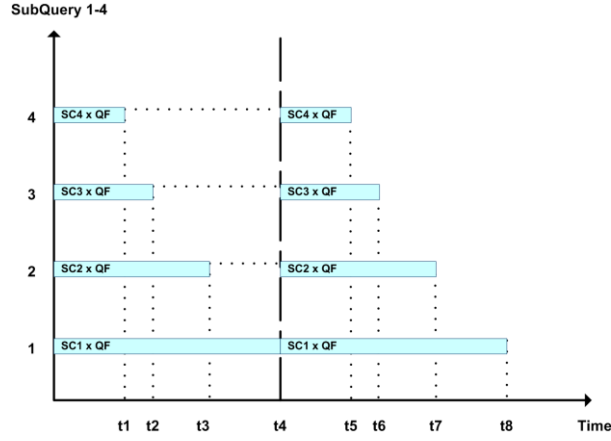


Figure 4.7 RGP Periods Time Gantt chart: (SC*S) per sub query

When all sub queries are completed a response message is generated. RGP periods are given in Figure 4.7. The figure illustrates a continuous network query which has four sub queries with different sample counts. Query instances are scheduled at t_4 , t_8 etc. In order to schedule next instance all sub query samples for the previous instance have to be collected. At this point a response message is posted and all sub queries are rescheduled by initializing their number of samples counts to their initial values.

4.8.1 Response Traffic Analysis

In SQS, a sensor network is formed for a limited period of time to process one network query. Sensors compile query readings at given frequencies. Response generation period is a function of Sample Count (SC) and Sampling Period (S) values received during setup. Each network query can be decomposed into maximum four sub-queries as defined in the setup packet. Compiled result of each sub query is locally buffered and one network packet is generated at Response Generation Periods (RGP) for all sub query responses.

$$RGP = \text{Max}(SC) * S \quad (4.1)$$

As the response generation method (RGM) may vary for each subquery $S(i)$, traffic intensity of the network as a function of network query type can be analyzed for the best and worst case sub query type combinations. For example: simple queries generate only one response at each node whereas complex queries may result in zero or one response.

If set of generation methods is defined as:

$$RGM - SET = \{ Simple, Complex \} \quad (4.2)$$

Then the response generation method for the network query can be given as follows:

$$NetQuery_{RGM} = \bigcup_{i=1}^4 SubQuery(i)_{RGM} - \{Complex\} \quad (4.3)$$

This predicate statement indicates that if at least one sub query is of Simple type then network query RGM is also Simple which generates one response per node. If all sub queries are Complex then the network query type is also Complex generating zero or one response per node.

Continuous queries cause periodic processing on the nodes. The number of periods or Query Cycle (QCycle) is found by dividing Query Time Period (QTP) by RGP from Equation (4.4).

$$QCycle = QTP / RGP \quad (4.4)$$

From eq. (4.4) and eq. (4.3), minimum and maximum number of packets injected into the network can be estimated for network query types. Table 4.5 illustrates the range of number of responses generated in an n-node query network.

Table 4.5 Traffic Intensity in N-hop sensor network

Query Type	Response Range
00(0,1)	[N, N]
01(0,1)	[0,N]
10(0,1)	[(N*QCycle), (N*QCycle)]
11(0,1)	[0, (N*QCycle)]

4.8.2 Response Message and Time Synchronization

A response message might carry a single node response or responses from multiple nodes. Thus, the response message is a variable length message as illustrated in Figure 4.8 and Response Count Field is used to tell the number of response messages contained in the packet. Response Data is the name of the data given to each single sensor's data which is depicted in Figure 4.9.

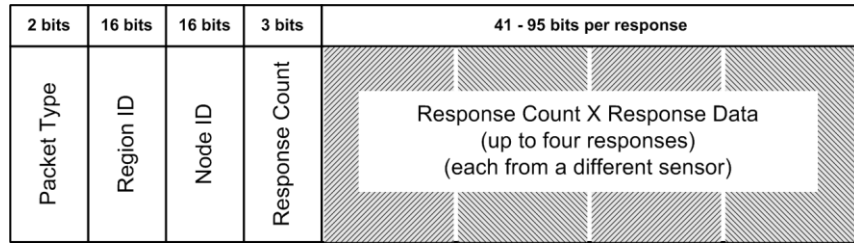


Figure 4.8 Response Packet

Each response data message contains a Hop Count, a Data Count and a Response Time field and Data Count times Sub query Context Data. These fields are used to uniquely identify each incoming response data at the driver node. In the proposed architecture, each response data carries its originator's hop count to specify how many hops away this event has occurred. This way, driver node can have a rough idea about the location of the event in terms of transmission range and hop count.

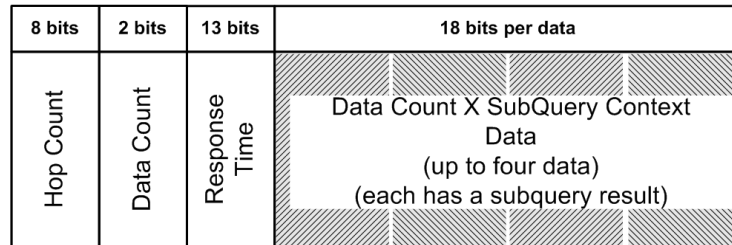


Figure 4.9 Response Data

Since not all of the nodes participating in query are required to match all sub queries, some nodes joining the query can be simple message forwarders. So, the Data count field is used to specify how many of the requested sub queries can a node answer to. The response of node follows response time field in the packet and is named as sub query context data which contains 16 bit response of the message and the index of the sub query this data refers to.

Sensor networks are mostly known as unreliable and there is no guarantee about the order of arrival of the events to the driver node. Therefore, response messages about the same event initiated by neighboring nodes might arrive at different times to the driver. While some packets might be lost on the way, others might arrive out of order or some nodes might run the reliable event delivery algorithm as a result of losing contact with their local parents. As a result, the driver node is expected to discriminate these messages from each other. In the case of a target tracking application, messages are required to be sorted by time to provide target path to the

monitoring application. In order to distinguish events from each other a timing mechanism is required.

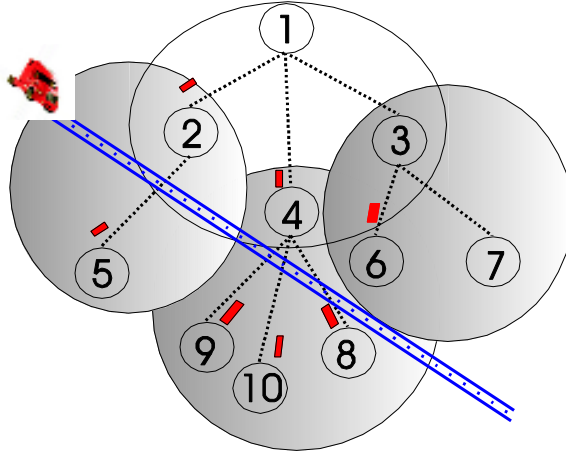


Figure 4.10 Timing Problem Demonstration

Recent research groups [29], [30] have used messaging based synchronization primitives but message transmission is the most costly operation on sensor nodes and it's not desirable for each node to transmit message to each other to keep synchronized. Proposed query processing algorithm provides its users with relative message delivery time so that client nodes can have a rough idea about the occurrence of event. Each response message has a response time field which is used for time stamping. Each time a response message is processed on each node at the forwarding path its time count is incremented by the amount of time difference between message reception and message transmission and by a theoretical propagation delay calculated by using 12kbps transmission throughput and packet length. Therefore, messages for the same event arriving via a longer path will have relatively greater time. Driver node can filter the events that have already been delivered to client node so that delayed packets are also taken care of.

In Figure 4.10 a sample scenario for timing problem is given. In this scenario several sensors have been deployed in an area to track any changes in the environment and the path of a truck is given. During the truck's movement, its movement is detected by 2,5,4,9,10,8 and 6 sensors respectively assuming that sensing and transmission ranges of sensors are same. However, when looked at the events from monitoring application point of view, it needs to discriminate the event's occurrence sequence so that it can have a rough idea about the path of the target. First, when the target enters the circled region, it will be detected by sensors 2 and 5. Second, it will be detected

by 4, 9, 8 and 10. Finally, it will be detected by 6. When the driver node processes the received messages, it will be able to figure out the truck's movement path.

4.8.3 Response Generation Method

Two kinds of response generation methods are supported by the architecture and Concatenation flag in the setup packet is used to switch between modes.

4.8.3.1 Immediate Message Delivery

For time critical applications like a fire in the forest or a chemical attack in the war, events should be delivered to driver nodes immediately. With concatenation flag unset, sensor nodes forward their messages to their parent nodes without delaying.

4.8.3.2 Aggregated Message Delivery

In order to save energy, messages that are carried over the broadcast network are concatenated at upper level nodes so that the number of transmitted packets is reduced. Since each packet has a header to identify the responding node, concatenation is more efficient in terms of number of bytes transmitted.

For example, each node which responds to a query message with one sub query will send a response message with 37 bits of message header for identifying message type, destination, region id and response count and 41 bits for response data, resulting in 78 bits. When the node is configured with the concatenation mode, it will put 4 messages with 41 bits of response data and a header to the packet instead of transmitting 78 bit response data four times resulting in 55% gain.

Whether nodes concatenate responses or not is configured by the driver node during setup. One drawback of concatenation is the latency for response messages. Each node waits for timeout period to transmit local data to see if anyone else is also transmitting data over itself so that it can concatenate with its own.

With concatenate flag set, sensors wait for a concat timeout interval to fill the packet with the responses from other nodes. Equation 4.5 gives timeout for a node to relay its message. A message can be transmitted prior to timeout if the packet is full at any time. Concatenation timeout is proportional to the level of the node in the tree, node's Neighbor Count (nc) and propagation time of the packet (T_{pg}).

$$T_{concat} = RGP.(fd - hc - 1). + Tpg * nc \quad (4.5)$$

Nodes in the lower levels of the tree wait less; whereas, nodes in the upper levels wait more. Each node also keeps track of received setup packets from the neighboring nodes which are correlated to the number of received unnecessary setup and response messages. Since RF is a broadcast medium, each receiving node in the transmission range will broadcast the packet again. Each setup packet receiving node will wait for a fixed duration to keep track of the number of nodes in the area to use as a feedback to make adjustments for concat timeout value.

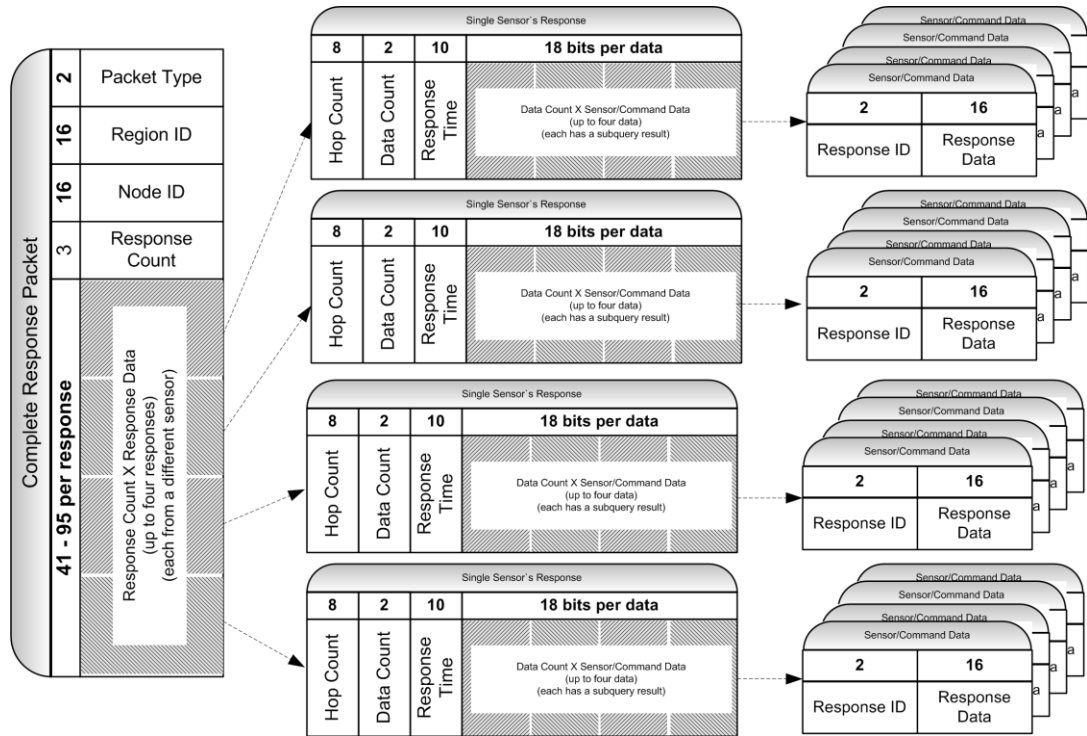


Figure 4.11 Response Packets in Detail

When concatenation timer expires, the node checks to see if there is any packet in the received buffer and transmits it. As the responses will gradually grow towards the sink, this metric considers that nodes at the higher levels of tree will wait more and nodes at the lower parts of the tree will wait less.

Figure 4.11 shows a typical concatenation process in terms of packaging. As it can be seen from, the architecture supports up to four messages that can be concatenated when a single sub query is used.

Table 4.6 Number of Sub queries vs. Concatenated Message Count

Number Of Subqueries	Concatenated Message Count
4	1
3	2
2	3
2	4

Three messages can be concatenated for two sub queries and two messages can be concatenated for three and four sub queries. It should be noted that since not all nodes can have the same abilities to respond to the sub queries, number of concatenated responses might vary according to the length of the response messages generated by nodes.

```

ProcessResponseMessage()
begin
  TOS Msg msg
  if (ReceiveMessage(&msg)) begin
    if EnoughPlaceExistsInLocalResponseMessage()
      CopyMessageToLocalResponse(msg)
    else begin
      CopyPartOfMessageUntilPacketIsFull(msg, local response message)
      TransmitMessage(local response message)
      AllocateBuffer(&local response message)
      CopyRemainingsOfMessage(msg, local response message)
    end
    if PacketFull(local response message)
      TransmitMessage(local response message)
    end

    if (local response message ready) begin
      if EnoughPlaceExistsInLocalResponseMessage()
        AppendLocalResponseToLocalResponseMessage()
      else begin
        TransmitMessage(local response message)
        AllocateBuffer(&local response message)
        RetryLocatingLocalResponse()
      end
      if PacketFull(local response message)
        TransmitMessage(local response message)
      end

      if (Expired(Concatenation timer)) begin
        if ThereIsResponseMessageNotSent()
          TransmitMessage(local response message)
        end
      end
    end
  end
end

```

Figure 4.12 Concatenation Algorithm

Concatenation algorithm is depicted in Figure 4.12. There are two sources of messages that can be concatenated into the same buffer. One of these is local response message which is generated by processing sub queries and another is received messages from other nodes. If a node receives a response message, it checks to see if there is space in concatenation buffer and appends the message to this area. Otherwise, it copies the part of message that can fit into the concatenation buffer and

transmits the concatenated buffer. Then, it allocates a new concatenation buffer and copies the remaining fields of the message to this area.

Similar things are done in the case of response generation. If the processing of all sub queries is completed, node appends its response message to concatenation buffer. If there isn't enough space, it transmits the concatenated packet and retries appending to a newly allocated buffer. After concatenation, if the buffer is full, it is also transmitted to not delay packet delivery for the next packet reception of response generation.

4.8.4 Response Delivery

Setup packets carry the query related parameters to the nodes and data packets are generated as a response. Data packets might include sensing related data as well as node related constants. The routing algorithm specifies two kinds of event delivery. These are end-to-end reliable event delivery and best effort event delivery.

With best effort event delivery option is selected, responses from the sensor nodes are assumed to be delivered to driver node without failure. Each node simply forwards data packets to its parent node. Since sensor lifetimes can be quite varying, topology failures may occur. In such environments, reliable event delivery can be achieved by transmission control schemes but the cost of retransmissions makes them unusable for sensor networks. In our architecture, end-to-end reliable event delivery is achieved by a dynamic-path-switching algorithm which makes each event data receiving node to take the responsibility of the delivery of the packet to its own parent node using ack packets and timeouts. End-to-end transmission is achieved between the last node receiving the event successfully on the return path and the sink node; not with the ultimate event source and the sink.

5 SOFTWARE ARCHITECTURE

SQS query processing system has been implemented on TinyOS [31] operating system for use in real sensor network scenarios. Therefore, the design of the software has been affected by operating system and programming language limitations. In this section, first the development tools and sensor hardware that were used are introduced and then the components of the SQS query processing system are described in detail in application framework subsection.

5.1 Development Platform

Mica2 and Mica2Dot hardware by Crossbow Inc [32], TinyOS operating system, Network Embedded Systems C (NesC) [33] programming language and Tiny Micro threading Operating System Simulator (TOSSIM) [34] simulator have been used during development. NesC is a subset of C language and TinyOS is a power efficient operating system with a small footprint. TOSSIM is a scalable simulator developed on top of TinyOS by making abstraction at the hardware level to PC components.

5.1.1 Sensor Mote Platform

As stated before Smart Dust project aims to build 1 cubic millimeter sensor nodes but currently these motes are readily available and research in this area is still in progress. In order to speed up the project several prototype motes have been designed by using commercial of the shelf (COTS) components in order to evaluate the behavior of a final Smart Dust mote. COTS Motes is an alternative means to test basic complete system behavior in a timely manner. COTS Motes has all the basic functionality of Smart Dust, but the devices are built in a tenth of the time. Instead of being a cubic millimeter in size, these devices are around a cubic inch in size (16,387 cubic millimeters). The COTS Mote serves as the current platform and runs a variety of algorithms that Smart Dust will run.

Several unstable and stable prototypes that use different technologies such as optical communication RF communication have been built. These prototypes were named as COTS Dust. The details and evaluation of the prototypes are given in [35].

The hardware designs in COTS Dust motes were followed by several other motes. WeC mote, which was designed in 09/1999, is the smallest in size of the COTS Dust motes and was designed with the original intent to test out algorithms based on messaging for distributed sensor networks.

Following the introduction of the WeC Mote, the beginnings of an operating system, TinyOS, for management of the different functions on board a Mote began development by UCB. The introduction of TinyOS initially consisted of a communication stack. Later Crossbow Inc [32] continued development of motes with Renee, Mica, Mica2 and Mica2Dot hardware. The technical specifications of each mote are given in Figure 5.1.




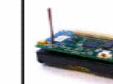
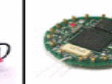
Mote Type	WeC	Renee	Mica	Mica2	Mica2Dot
					
Microcontroller					
Type	AT90LS8535	Atmega163	Atmega128	Atmega128	Atmega128
CPU Clock (Mhz)	4	4	4	7.3827	4
Program Memory (KB)	8	16	128	128	128
Ram (KB)	0.5	1	4	4	4
UARTs	1	1	2 (only 1 used)	2	2
SPI	1	1	1	1	1
I2C	Software	Software	Software	Hardware	Hardware
Nonvolatile storage					
Chip	24LC256		AT45DB041B		
Size (KB)	32		512		
Radio Communication					
Radio	RFM TR1000			Chipcon CC1000	
Frequency	916 (single freq)			916/433 (multiple channels)	
Radio speed (kbps)	OOK		ASK	FSK	
Transmit Power Control	Programmable resistor potentiometer			Programmable via CC1000 registers	
Encoding	SecDed (software)			Manchester (hardware)	

Figure 5.1 COTS Motes

Mica2 and Mica2Dot are currently the most advanced versions of the COTS motes. The most important advantage of these motes is they have lower power consumption. The radio interface has been replaced with a better chip that supports better noise immunity, better range and different power down modes. The power breakdown of the mote components can be found in mote user's manual [36]. According to this document when the system components are used effectively with an AA battery system lifetime can be up to 17.35 months. The energy dissipation of certain operations is as follows:

Table 5.1 The Energy Dissipation of Operations for Mica2

Operation	Energy Dissipation
Flash memory write access	15mA
Radio transmit	12mA
Radio receive	8mA
CPU running in full duty cycle	8mA
Sensor board data acquisition	5mA

The radio on the Mica2 and Mica2Dot can be adjusted for a range of output power levels. When output power is set to maximum, a transmission range of 30 meters is achieved.

Mica2 has an extensible interface for plugging different sensor boards using a 51 pin connector. Several sensor boards [36] are available from Crossbow Inc. Available sensing interfaces for the motes include light, temperature, acceleration, magnetometer, microphone, tone detector and ultrasonic sensors. Acceleration sensor is generally used to calculate the acceleration of a moving target in x and y coordinates relative to the node. Magnetometer is used for detection of targets. Different objects have different magnetic characteristics and when the read values are combined with signal processing detection of a person is achievable in an area. Ultrasonic sensor is a range detection sensor can get the position of an object relative to sensor node up to 2.5 meters range with 6 cm accuracy.

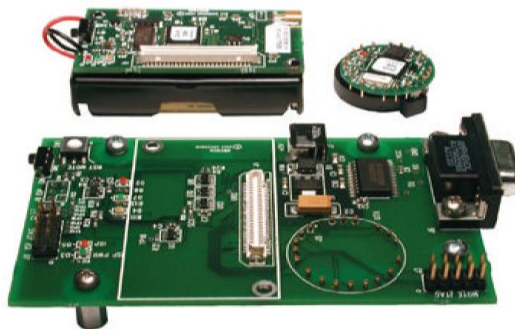


Figure 5.2 Mote Programming Interface

Mica2 and Mica2Dot are programmed by using Mote Programming Interface which is shown in Figure 5.2. Mote programming interface is also used to collect data from the sensor network to the pc. This is achieved by plugging a mote to the sensor board and instructing sensor to forward the received messages from the RF to the PC serial port. This mote is usually named base station in applications.

5.1.2 TINYOS

Existing operating systems don't meet requirements of Wireless Sensor Networks since they were designed for more complex systems such personal computers. A WSN operating system must be efficient in terms of memory, processor, and power so that it meets strict application requirements. The extreme constraints of sensor nodes make it impractical to use legacy systems. Two issues must be addressed by the operating system: sensor nodes are concurrency intensive - several different

flows of data must be kept moving simultaneously, and the system must provide efficient modularity - hardware specific and application specific components must snap together with little processing and storage overhead.

5.1.2.1 TinyOS Execution Model: Event Based Execution

TinyOS maintains a two-level scheduling structure as shown in Figure 5.3, so a small amount of processing associated with hardware events can be performed immediately while long running tasks are interrupted. The execution model is similar to finite state machine models, but considerably more programmable.

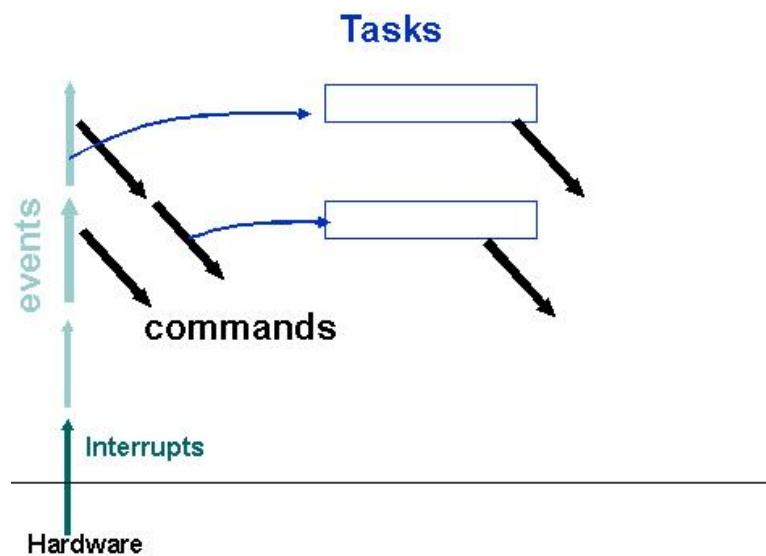


Figure 5.3 TinyOS Execution Model

To provide the extreme levels of operating efficiency required in wireless sensor networks, TinyOS uses event based execution. A thread based approach requires that stack space be pre-allocated for each execution context. Additionally, the context switch overhead of threaded systems is significantly higher than those of an event-base system.

In TinyOS, a single execution context is shared between unrelated processing tasks. When an event arrives, it brings the required execution context with it. When the event processing is completed, it is returned back to the system. A key to limiting power consumption is to identify when there is no useful work to be performed and to enter an ultra-low power state. In TinyOS, all tasks associated with an event are handled rapidly after each event is signaled. When an event and all tasks are fully processed, unused CPU cycles are spent in the sleep state as opposed to actively

looking for the next interesting event. Eliminating blocking and polling prevents unnecessary CPU activity.

Figure 5.3 shows how TinyOS code execution model performs. In normal condition CPU is in low power state and is awakened by a hardware interrupt which calls the related event function. Event function triggers the related tasks in the system and they are executed until completion. During this time if another event is received TASKS are interrupted and events are serviced.

A limiting factor of TinyOS is that long-running calculations can disrupt the execution of other time critical subsystems. Therefore, task based execution model is employed with a task queue which is used to queue pending tasks. The important factor is that programmers should employ state based programming for long executing operations. So that during task switch other tasks pending in the queue can be serviced.

The need for synchronization is also seen in sensor network applications but a lightweight concurrency model is proposed for atomic operations instead of complex spinning semaphores. In TinyOS, code that is executed inside of a task is guaranteed to run to completion without being interrupted by other tasks. To guard tasks against reentrant interrupt codes, atomic operations are introduced by NesC language which disables interrupts to allow critical sections to be created.

5.1.2.2 TinyOS Component Model

Since storage area for code is limited, TinyOS includes a specially designed component model targeting highly efficient modularity and easy composition. The component model allows an application developer to be able to easily combine independent components into an application specific configuration.

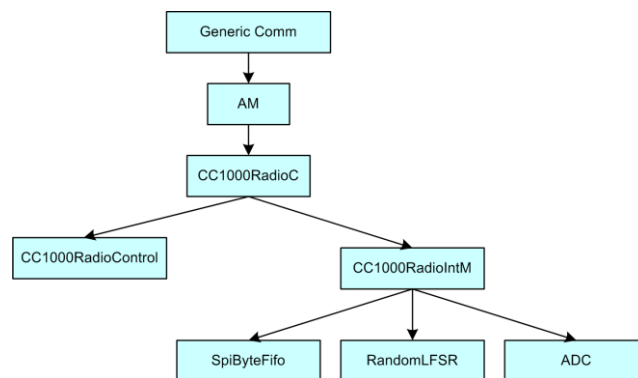


Figure 5.4 Component Graph of GenericComm

In TinyOS, each module is defined by the set of commands and events that makes up its interface. In turn, a complete system specification is a listing of the components to include plus a specification for the interconnection between components. The TinyOS component has four interrelated parts: a set of command handlers, a set of event handlers, an encapsulated private data frame, and a bundle of simple tasks. Tasks, commands, and event handlers execute in the context of the frame and operate on its state. To facilitate modularity, each component also declares the commands it uses and the events it signals. Event handlers are invoked to deal with hardware events, either directly or indirectly.

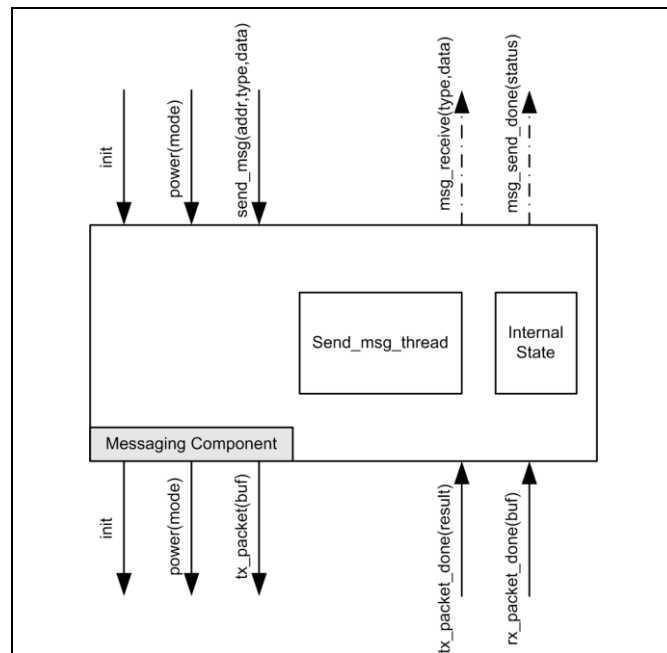


Figure 5.5 A sample component: Messaging Component.

Tasks perform the primary work in a TinyOS application. They are atomic with respect to other tasks and run to completion, though they can be preempted by events. Tasks can call lower level commands, signal higher level events, and schedule other tasks within a component. The run-to-completion semantics of tasks make it possible to allocate a single stack that is assigned to the currently executing task. This is essential in memory constrained systems.

Messaging component is illustrated in Figure 5.5. Generally programming steps in TinyOS is as follows. First, a programmer designs the component prior to coding and inputs and outputs of a component and their logical relationships are specified. Second, the description of the component is specified in configuration file and lastly

implementation is done in module file with tasks. In Figure 5.5, a sample design is given.

5.1.2.3 AM Communication Paradigm

Active Messages communication model is the primary building blocks for networking in TinyOS. Active Messages (AM) is a simple, extensible paradigm for message-based communication widely used in parallel and distributed computing systems [37]. Each Active Message contains the name of an application-level handler to be invoked on a target node upon arrival and a data payload to pass in as arguments. The handler function serves the dual purpose of extracting the message from the network and either integrating the data into the computation or sending a response message. The network is modeled as a pipeline with minimal buffering for messages. This eliminates many of the buffering difficulties faced by communication schemes that use blocking protocols or special send/receive buffers. To prevent network congestion and ensure adequate performance, message handlers must be able to execute quickly and asynchronously.

Dest	AM Type	Group	Length	Data	CRC
2 bytes	1 byte	1 byte	1 byte	29 bytes	2 bytes

Figure 5.6 TinyOS Packet Format

The event based handler invocation model allows application developers to avoid busy-waiting for data to arrive and allows the system to overlap communication with their activities such as interacting with sensors or executing other applications. It is this event centric nature of Active Messages which makes it a natural fit for TinyOS.

TinyOS Packet format is given in Figure 5.6. Address of the destination is set in *Dest* field and the application number that this message belongs to is set in *AM type* section of the message. Sensor nodes can be communicating with multicast scheme; therefore, group of the recipients of this message is set in *Group* field. Default group ID is 0x7f and user data is put in *Data* section of the message and *Length* of the message is set by the sender to the length of the data requested to be transmitted. Underlying communication scheme computes two bytes *Cyclic Redundancy Check* (CRC) and transmits the packet.

Another issue fundamental to the TinyOS communication system is its storage model. As data arrives over the radio, it must be stored into a memory buffer. The active messages dispatch layer then delivers the buffer up to the applications. In many cases, applications will wish to keep the message buffers that are delivered to them. In the case of the multi-hop communication, applications would need to keep the buffer long enough for it to be transmitted to the next node. To handle this, TinyOS requires that each application return an unused message buffer to the radio subsystem each time a message is delivered. The radio simply maintains one extra buffer to receive the next message into. After reception the buffer is transferred up to the application. Then a free buffer is handed back by the application component to the radio system. This free buffer is then filled by the next message.

5.1.3 NesC

NesC is an extension to C [38] designed to embody the structuring concepts and execution model of TinyOS [31]. NesC supports a programming model that integrates reactivity to the environment, concurrency, and communication. By performing whole-program optimizations and compile-time data race detection, nesC simplifies application development, reduces code size, and eliminates many sources of potential bugs.

Mote applications are deeply tied to hardware, and each mote runs a single application at a time. This approach yields three important properties. First, all resources are known statically. Second, rather than employing a general-purpose OS, applications are built from a suite of reusable system components coupled with application-specific code. Third, the hardware/software boundary varies depending on the application and hardware platform; it is important to design for flexible decomposition.

The basic concepts behind nesC are:

Separation of construction and composition: programs are built out of components, which are assembled (“wired”) to form whole programs. Components define two scopes, one for their specification (containing the names of their interface instances) and one for their implementation. Components have internal concurrency in the form of tasks. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt.

Specification of component behavior in terms of set of interfaces: Interfaces may be provided or used by the component. The provided interfaces are intended to represent the functionality that the component provides to its user; the used interfaces represent the functionality the component needs to perform its job.

Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider (commands) and a set to be implemented by the interface's user (events). This allows a single interface to represent a complex interaction between components (e.g., Registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the *sendDone* event. Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts.

Components are statically linked to each other via their interfaces: This increases runtime efficiency, encourages robust design, and allows for better static analysis of programs.

The concurrency model of nesC: It is based on run-to-completion tasks, and interrupts handlers which may interrupt tasks and each other: The nesC compiler signals the potential data races caused by the interrupt handlers.

Whole-program compilation: NesC is designed under the expectation that code will be generated by whole-program compilers. This allows for better code generation and analysis. An example of this is nesC's compile-time data race detector.

5.1.3.1 Component Specification

NesC applications are built by writing and assembling components. A component *provides* and *uses* interfaces. These interfaces are the only point of access to the component. An interface generally models some service and is specified by an interface type.

<pre> configuration TimerC { provides interface Timer[uint8_t id]; provides interface StdControl; } implementation { components TimerM, ClockC, NoLeds; components HPLPowerManagementM; TimerM.Leds -> NoLeds; TimerM.Clock -> ClockC; TimerM.PowerManagement -> HPLPowerManagementM; StdControl = TimerM; Timer = TimerM; } </pre>	<pre> module TimerM { provides interface Timer[uint8_t id]; provides interface StdControl; uses { interface Leds; interface Clock; interface PowerManagement; } } </pre>
---	---

Figure 5.7 Timer Module Configuration

The configuration of Timer component is given in Figure 5.7 and logical relationship between components are depicted in Figure 5.8. Configuration states that timer component provides an array of Timer and a StdControl interface to the user while it implements its Clock interface by wiring it to ClockC component, PowerManagement interface to HPLPowerManagementM component and Leds to NoLeds interface. Leds, Clock and PowerManagement are used by this component; whereas, Timer and StdControl are provided.

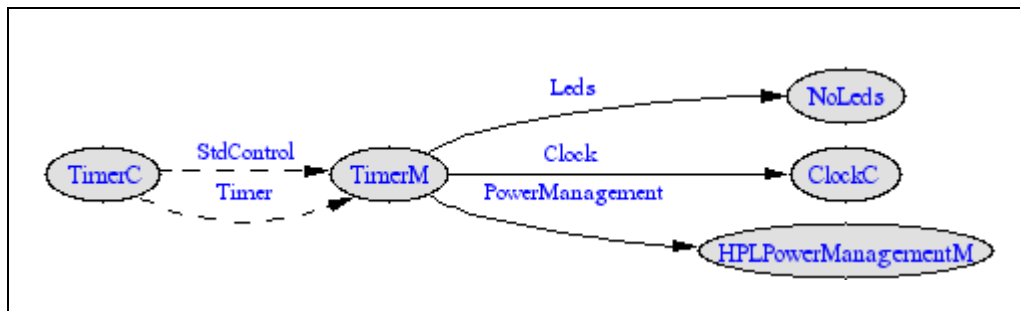


Figure 5.8 Relationship diagram of Timer Module

Interfaces in nesC are bidirectional: they contain *commands* and *events*, both of which are essentially functions. The providers or an interface implement the commands, while the users' implements the events. In Figure 5.9, the definition of Timer interface and StdControl interface are given. StdControl interface is usually used for initializing a component. A component user uses init function to initialize the component and start function to start running or stop function to stop. If a component requires extra services to be represented to user, it defines its own

interface and provides it to user additionally. For instance, Timer component provides StdControl interface for initializing and Timer interface for accessing timer services. Start and stop commands are implemented in Timer module and fired event is provided to user to run when a timer tick event occurs. Events are generally caused by hardware interrupts; therefore, it's crucial that user provided functions be implemented in state machine transitions instead of a monolithic approach.

<pre> interface StdControl { command result_t init(); command result_t start(); command result_t stop(); } </pre>	<pre> interface Timer { command result_t start(char type, uint32_t interval); command result_t stop(); event result_t fired(); } </pre>
---	--

Figure 5.9 Sample Interfaces

The separation of interface type definitions from their use in components promotes the definition of standard interfaces, making components more reusable and flexible. A component can provide and use the same interface type, or provide the same interface multiple times. In these cases, the component must give each interface instance a different name using the “as” notation.

5.1.3.2 Component Implementation

There are two types of components in nesC: modules and configurations. Modules provide application code, implementing one or more interfaces. Configurations are used to wire other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components used.

The body of a module is written in C-like code, with straightforward extensions such as call atomic keyword is used to run a command and signal keyword is used to trigger and event function provided to user.

The explicit wiring of components via interfaces, combined with the removal of function pointer types, makes the control-flow between components explicit. Module variables are private. This makes it much easier to write correct components.

5.1.3.3 Concurrency and Atomicity

Concurrency is central to nesC components: events (and commands) may be signaled directly or indirectly by an interrupt, which makes them asynchronous code. NesC provides atomic tasks and events to handle these situations.

Data races occur due to concurrent updates to shared state by unmasked interrupts and tasks. For instance, a task accessing a local variable can be interrupted and interrupt service routine or events can update the same variable. Because of simplicity of the sensor network applications compile time concurrency checks are applied by nesC. In order to prevent them, a compiler must understand the concurrency model and determine the target of every update.

In TinyOS, code runs either asynchronously in response to an interrupt or in a synchronously scheduled task. NesC categorizes code as asynchronous and synchronous. Asynchronous codes are the ones that can be executed by an event and synchronous codes are the ones that are executable by a task. Tasks run to completion in TinyOS unless interrupted. Therefore, Synchronous Code is atomic with respect to other Synchronous Code.

“Atomic” keyword with parenthesis is used to enclose code accessing shared variable and during this time interrupts are disabled to prevent accessing of data at the same time. If the user does not use atomic keyword with his code, compiler detects data races by comparing the variables accessed by synchronous and asynchronous codes.

```
Module Test{
  Implementation{
    bool busy;
    TOS_Msg m;
    ...
    event result_t MsgSend.sendDone(TOS_MsgPtr msg,result_t success)
    {
      atomic
      {
        busy = FALSE;
      }
    }

    event result_t Timer.fired()
    {
      bool localBusy;
      atomic {
        localBusy = busy;
        busy = TRUE;
      }
      if (!localBusy)
      {
        bool ret;
        ret = call MsgSend.send(TOS_UART_ADDR,sizeof(int),&m);
        if (ret)
        {
          atomic{
            busy = TRUE;
          }
        }
      }
      return SUCCESS;
    }
  }
}
```

Figure 5.10 Atomic Usage

In Figure 5.10, a code snippet of test application has been provided. This application transmits data with each timer tick and checks to see if previous transmission is still in progress. Busy local variable has been used to synchronize each function and atomic keyword is used while accessing busy variable.

Currently, atomic is implemented by enabling and disabling interrupts which has a very low overhead. However, leaving interrupts disabled for a long period delays interrupt handling, which makes the system less responsive. To minimize this effect, atomic statements are not allowed to call commands or signal events, either directly or in a called function.

5.1.4 SerialForwarder

SerialForwarder application in Figure 5.11 is used to interact with simulation or real motes. Mica2 and Mica2dot hardware is connected to a PC through serial port that's why application is named SerialForwarder. The application serves as a forwarder between selectable input controls and it redirects read data to the clients connected to a TCP/IP socket server. As a result, mote data is stored in a TCP server which can be accessed by external application such as Java applications converted using MIG application.

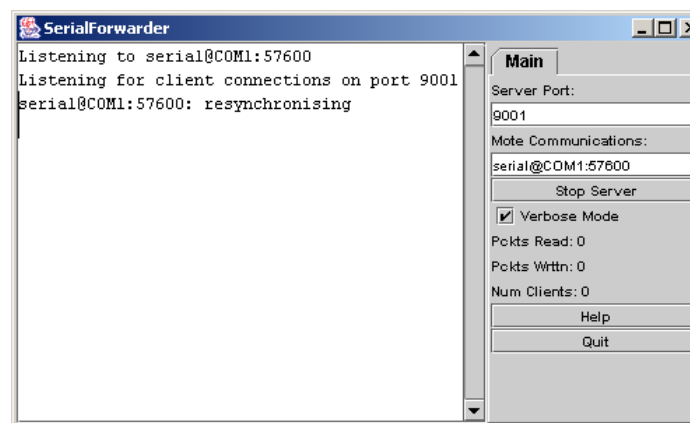


Figure 5.11 SerialForwarder

TOSSIM Serial input control is used to retrieve data from simulation environment instead of real motes. When the application is started in TOSSIM Snoop mode all messages transmitted in the simulation environment is forwarded to SerialForwarder regardless of the message destination. Serial Port is used to retrieve data from the PC's serial port.

Currently, SerialForwarder application has been depreciated and renamed as old SerialForwarder because of the lack of acknowledge messages in the data exchange between TCP clients. Either new SerialForward application with framed and acknowledged messages or old SerialForwarder with no acknowledgements and frames can be used to retrieve data.

5.1.5 TOSSIM

TOSSIM is a discrete event simulator for TinyOS sensor networks. Instead of compiling a TinyOS application for a mote, users can compile it into the TOSSIM framework, which runs on a PC. This allows users to debug, test, and analyze algorithms in a controlled and repeatable environment. As TOSSIM runs on a PC, users can examine their TinyOS code using debuggers and other development tools.

The simulator engine provides a set of communication services for interacting with external applications which is shown in Figure 5.12. These services allow programs to connect to TOSSIM over a TCP socket to monitor or actuate a running simulation. Details of the ADC and radio models, such as readings and loss rates, can be both queried and set. Programs can also receive higher level information, such as packet transmissions and receptions or application-level events.

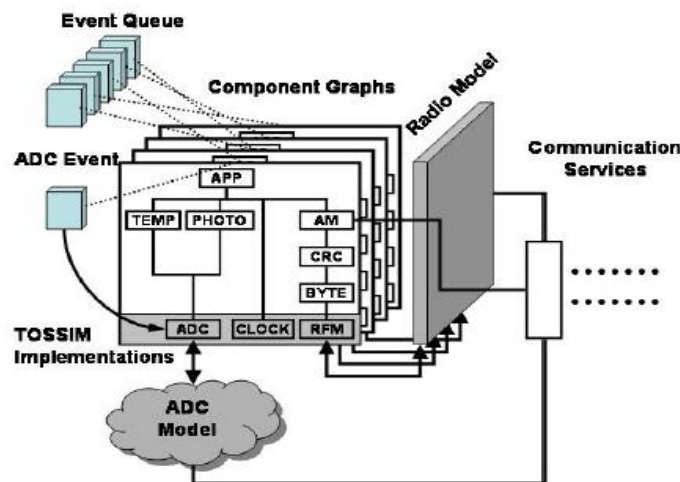


Figure 5.12 TOSSIM Architecture

While TOSSIM can be used to understand the causes of behavior observed in the real world, it does not capture all of them such as preemptive interrupts and harsh conditions. The flow diagram of TOSSIM is given in Figure 5.13. This figure shows

that tasks are run to completion in simulator as in real world nodes but they are not interrupted by events. After completion of tasks, event queue is checked to see if there is any pending interrupt function in the queue and if there is one, it is executed.

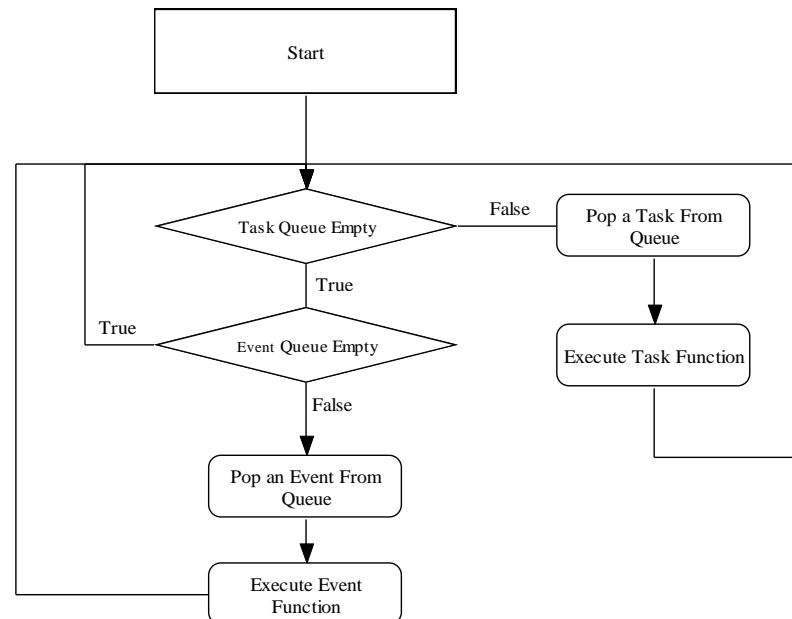


Figure 5.13 Flow Diagram of TOSSIM

Although TOSSIM can be used to simulate most of the real world events, it makes several assumptions:

TOSSIM captures TinyOS' behavior at a very low level: It simulates the network at the bit level, simulates each individual ADC capture, and every interrupt in the system.

Non-preemptive mode of operation: While TOSSIM precisely times interrupts (allowing things like bit-level radio simulation), it does not model execution time. From TOSSIM's perspective, a piece of code runs instantaneously and can not be interrupted.

Time is kept at a 4MHz granularity: (the CPU clock rate of the rene and mica platforms). This also means that spin locks or task spin locks will never exit: as the code runs instantaneously, the event that would allow the spin to stop will not occur until the code completes.

TOSSIM tries to model communication behavior: TOSSIM itself does not model the real world; however, it provides models to allow an external utility to simulate real world conditions. TOSSIM does not model radio propagation; instead, it

provides a radio abstraction of directed independent bit errors between two nodes. An external program can provide a desired radio model and map it to these bit errors. Having directed bit error rates means that asymmetric links can be easily modeled. Independent bit errors mean longer packets have a higher probability of corruption, and each packet's loss probability is independent. Currently, TOSSIM simulates the 40Kbit Radio Frequency Model (RFM) mica networking stack, including the MAC, encoding, timing, and synchronous acknowledgements. While Mica2 stack is better it is not included in the simulation.

TOSSIM does not model power draw or energy consumption: However, it is very simple to add annotations to components that consume power to provide information on when their power states change (e.g., turned on or turned off). After a simulation is run, a user can apply an energy or power model to these transitions, calculating overall energy consumption. Because TOSSIM does not model CPU execution time, it cannot easily provide accurate information for calculating CPU energy consumption.

TOSSIM builds directly from TinyOS code: To simulate a protocol or system, TinyOS implementation of it has to be written. This allows simulated code to be executed on real motes as well

```
$ build/pc/main.exe -r=lossy 10
SIM: Initializing sockets
SIM: Created server socket listening on port 10584.
SIM: Created server socket listening on port 10585.
SIM: clientAcceptThread running.
SIM: commandReadThread running.
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
0 0:0:0.00000000: initial battery :3000.000000
SIM: spatial model initialized.
SIM: RFM model initialized at 40 kbit/sec.
Initializing lossy model from lossy.nss.
0 0:0:0.00000000: SIM RADIO: 2 -> 9 : 0.000010 neighbor_count = 1
0 0:0:0.00000000: SIM RADIO: 2 -> 10 : 0.000010 neighbor_count = 2
0 0:0:0.00000000: SIM RADIO: 6 -> 9 : 0.000010 neighbor_count = 1
0 0:0:0.00000000: SIM RADIO: 6 -> 10 : 0.000010 neighbor_count = 2
0 0:0:0.00000000: SIM RADIO: 7 -> 8 : 0.000010 neighbor_count = 1
0 0:0:0.00000000: SIM RADIO: 8 -> 7 : 0.000010 neighbor_count = 1
0 0:0:0.00000000: SIM RADIO: 9 -> 2 : 0.000010 neighbor_count = 1
0 0:0:0.00000000: SIM RADIO: 9 -> 6 : 0.000010 neighbor_count = 2
0 0:0:0.00000000: SIM RADIO: 9 -> 10 : 0.000010 neighbor_count = 3
0 0:0:0.00000000: SIM RADIO: 10 -> 2 : 0.000010 neighbor_count = 1
```

Figure 5.14 A Sample Run of TOSSIM

A sample run snapshot of TOSSIM is given in Figure 5.14. Simulation starts with socket initialization for external events and then battery levels of each mote is set. Later, connectivity graph for the simulation is formed.

5.1.5.1 Network Monitoring and Packet Injection

To interact with a simulated network, SerialForwarder application has to be used. To work with TOSSIM, SerialForwarder's input source must be set appropriately. TOSSIM provides two modes: communication through a serial port to mote 0 and network snooping.

The serial port mode interacts with mote 0 over its serial port. Programs connecting to SerialForwarder can read messages mote 0 sends to its serial port, and send messages to mote 0 over its serial port. The snooping mode sits on top of the TOSSIM network model. Programs connecting to SerialForwarder hear every radio message sent in the network, and can inject radio messages to arrive (without error) at any mote. Because this mode outputs every message sent, it does not consider loss; programs connecting will hear packets that might not arrive successfully at any mote.

Message Interface Generator (MIG) is a tool that generates Java classes for TinyOS packets which can also be used to interact with motes. The MIG tool parses C structures in the application for TinyOS packets and builds a Java class with accessors for each of the packet fields. The message classes can also unpack from and pack to byte arrays.

5.1.5.2 Radio Models

TOSSIM simulates the TinyOS network at the bit level, using TinyOS component implementations almost identical to the mica 40Kbit RFM-based stack. TOSSIM provides two radio models: simple and lossy. The mica2 CC1000-based stack does not currently have a simulation implementation.

In TOSSIM, a network signal is either a one or zero. All signals are of equal strength, and collision is modeled as a logical or; there is no cancellation. This means that distance does not affect signal strength; if mote B is very close to mote A, it cannot cut through the signal from far-away mote C. This makes interference in TOSSIM generally worse than expected real world behavior.

The “simple” radio model places all nodes in a single cell. Every bit transmitted is received without error. Although no bits are corrupted due to error, two motes can transmit at the same time; every mote in the cell will hear the overlap of the signals, which will almost certainly be a corrupted packet.

The simple model is useful for testing single-hop algorithms and TinyOS components for correctness. Deterministic packet reception allows deterministic results. The “lossy” radio model places the nodes in a directed graph. Each edge (a, b) in the graph means a’s signal can be heard by b. Every edge has a value in the range (0, 1), representing the probability a bit sent by a will be corrupted (flipped) when b hears it. For example, a value of 0.01 means each bit transmitted has a 1% chance of being flipped, while 1.0 means every bit will be flipped and 0.0 means bits will be transmitted without error. Each bit is considered independently. However, having 0 loss rate doesn’t mean that packets won’t get corrupted. Two nodes transmitting at the same time cause each others data to be overrun although link is lossless.

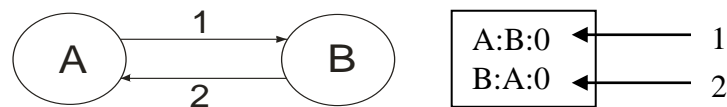


Figure 5.15 Lossy Model Configuration

By making the graph directed, TOSSIM can model asymmetric links, which is suggested by initial empirical studies as a common occurrence in sensor networks. By specifying error at the bit level, TOSSIM can capture many causes of packet loss and noise in a TinyOS network, including missed start symbols, data corruption, and acknowledgement errors. The lossy model models interference and corruption, but it does not model noise; if no mote transmits, every mote will hear a perfectly clear channel.

5.1.5.3 ADC Models

ADC is used to access sensing interface and gather data. TOSSIM provides two ADC models as random and generic to simulate data gathered from ADC. Whenever any channel in the ADC is sampled in the random model, it returns a 10-bit random value (the rene and mica ADCs are 10 bits).

The general model also provides random values by default, but has added functionality. Just as external applications can actuate the lossy network model, they

can also actuate the generic ADC model using the TOSSIM control channel, setting the value for any ADC port on any mote.

5.1.5.4 TinyViz

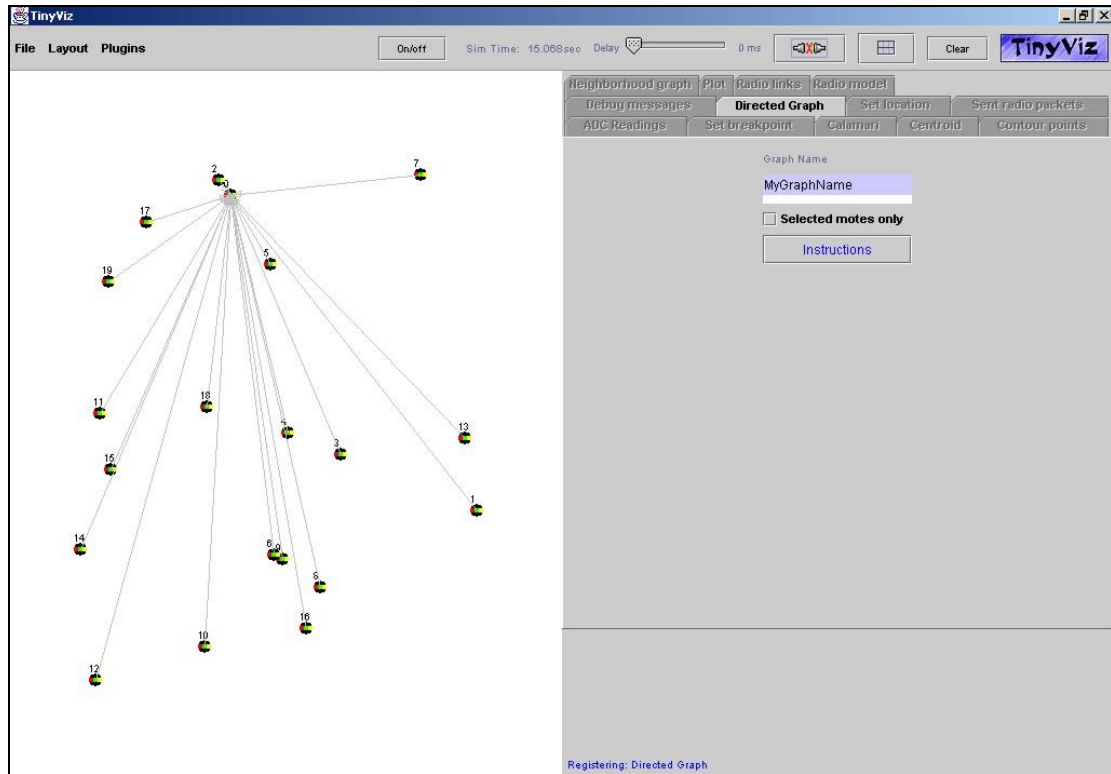


Figure 5.16 TinyViz

TinyViz as seen in Figure 5.16 is a Java visualization and actuation environment for TOSSIM. TinyViz can be attached to a running simulation. This allows users to be sure that TinyViz captures all of the events in a given simulation. TinyViz is not actually a visualizer; instead, it is a framework in which plugins can provide desired functionality. The TinyViz engine uses an event-driven model, which allows easy mapping between TinyOS's event-based execution and event-driven GUIs. By itself, the application does very little; drop-in plugins provide user functionality. TinyViz has an event bus, which reads events from a simulation and publishes them to all active plugins.

Users can write new plugins, which TinyViz can dynamically load. A simple event bus sits in the center of TinyViz; simulator messages sent to TinyViz appear as events, which any plugin can respond to. For example, when a mote transmits a packet in TOSSIM, the simulator sends a packet send message to TinyViz, which generates a packet send event and broadcasts it on the event bus. A networking

plugin can listen for packet send events and update TinyViz node state and draw an animation of the communication.

5.1.5.5 Concurrency Model

TOSSIM captures the TinyOS event-driven concurrency model at interrupt and task granularity. TOSSIM models each TinyOS interrupt as a simulation event. Each event is associated with a specific mote. Simulator events run atomically with respect to one another. Therefore, unlike on real hardware, interrupts cannot pre-empt one another. After each simulator event executes, TOSSIM checks the task queue for any pending tasks, and executes all of them in FIFO scheduling order. In TOSSIM, interrupts do not pre-empt tasks. This method of task execution means that spinning tasks (e.g., tasks that enqueue themselves in a spin-lock fashion) will cause TOSSIM to execute that task indefinitely. Using tasks in this way is contrary to the event-driven TinyOS programming model. Once a simulator event calls an interrupt handler, the handler executes TinyOS code through commands and events.

5.2 Application Framework

SQS query processing system consists of several applications from the lowest tier to the middleware. The aim of these tools is to bring sensor network capabilities to a form that is easily monitored and controlled by the user. Overall view of application framework is given in Figure 5.17.

The components of the application suite are:

SeMA Sensor Protocol: SeMA sensor protocol is the core facility of the system. It provides necessary low level coding in NesC language on top of TinyOS to achieve multi hop communication between the nodes, sensing interface access for data gathering and parsing of network packets in order to disseminate user driven queries and further process and reply to the requests. Details of query processing and dissemination algorithms are given in Chapter 4 of the thesis.

Driver: Currently, driver which is to implement driver capability of the architecture acts as an abstraction layer on top of TinyOS operating system so that any user can retrieve data using TCP/IP sockets.

Querier: Querier application is used to inject user queries and retrieve the results.

Oscilloscope: Oscilloscope application is used to monitor the change characteristics of sensor readings in time.

Database Server: Retrieved query results are stored in a database server as three tables so that results can be further processed using data mining tools.

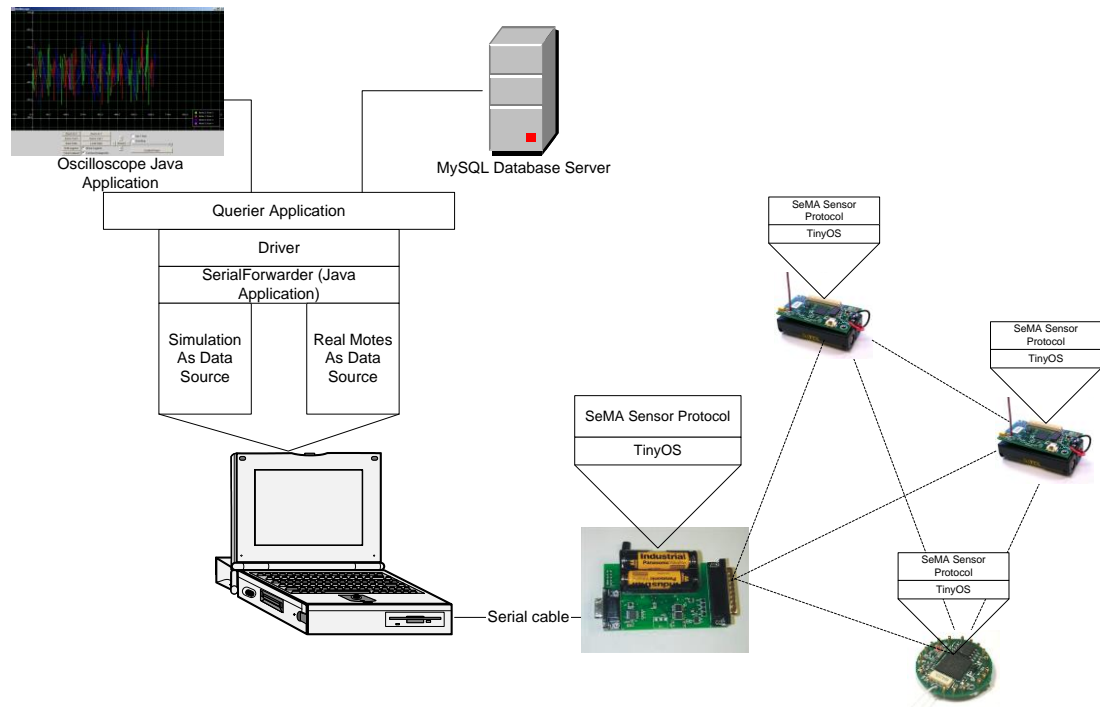


Figure 5.17 SQS Application Framework

A general usage scenario is as follows: User injects a query as “retrieve average daily temperature after taking 1 sample at each hour for the next 7 days” using querier application. Querier application converts user requested data to bit coded sensor network packet format and sends the application to driver application. Driver application encapsulates the received packet with an HDLC like framing protocol with acknowledgements to achieve reliable packet delivery between the mote and PC and sends the packet from the serial port using SerialForwarder application. The mote connected to the programmer board and identified as 0, broadcasts the request using RF interface and starts processing the request. Other motes in the network receive query request from the RF interface and record ID of the received message as parent ID and send their response to the mote with this ID every sampling period. Messages received from the sensor network via serial port are encoded with framing again. Driver application strips off the frames and makes necessary acknowledgements and posts the received message to the querier application. Querier

application displays the message in a listbox and it also posts the received results in Oscilloscope application's packet format to the oscilloscope application. User monitors the result from the oscilloscope application for different hop counts of the network.

5.2.1 SeMA Sensor Protocol Application

As most of the TinyOS applications SeMA sensor protocol application is created with the design of the wiring of the component. Configuration file is used to wire application requested interfaces to actual implementations. Since the software has been developed for multiple platforms including Mica2, Mica2dot and PC (for simulation) system dependent definitions are defined in "smtypes.h" header file. Depending on the definitions component linkage is done in the configuration file "smRouter.nc". Current hardware configuration and definitions are listed in Figure 5.18. As new sensor hardware become available to the public, configuration of the interfaces has to be done in "smtypes.h" file.

```
#if defined PLATFORM_MICA2
    #define HAVE_VOLTAGE_SENSOR          1
    #define HAVE_PHOTO_SENSOR           1
    #define HAVE_TEMP_SENSOR            1
    #define HAVE_RADIO_CONTROL          1
    // #define HAVE_SYSTIME              1
    #define HAVE_NETWORK_MON            1
#endif

#if defined PLATFORM_PC
    #define HAVE_PHOTO_SENSOR           1
    #define HAVE_TEMP_SENSOR            1
    #define HAVE_NETWORK_MON            1
    #define SEMATERMINATE               1
    #define SEMA_TOSSIM_EXTENSIONS      1
#endif

#if defined PLATFORM_MICA2DOT
    #define HAVE_TEMP_SENSOR            1
    #define HAVE_RADIO_CONTROL          1
    // #define HAVE_SYSTIME              1
    #define HAVE_NETWORK_MON            1
#endif
```

Figure 5.18 Hardware Dependent Definitions

Mica2 and Mica2dot hardware have voltage, temperature sensors and additionally Mica2 has photo sensor. Radio control definition is used to control the transmission range of the communication interface. Transmission range can be configured dynamically during system runtime or initially. Currently, since transmission range test have not been completed interfaces relating to hardware register access modules are defined but not called in the code. SysTime definition is used to access modules that allow system timer snapshots during runtime so that any time difference between

two events can be calculated. However, Mica2 and Mica2dot hardware have 16 bit CPU and SysTime interface provides user with times in terms of microseconds. Therefore, maximum time difference that can be measured using this interface is 65 milliseconds. As a result of our system's greater time difference calculation requirement SysTime interface is currently disabled. Network monitoring definition is required if the user wants to include logical topology monitoring of the network during runtime using Monitoring Tool for Networked Sensors (MoTeS) [39] application. Several modifications to the simulator have been made in order to automate running of multiple tests. These modifications are enabled by including HAVE_TOSSIM_EXTENSIONS in the hardware configuration. These modifications have been made on TinyOS 1.1 release and require simulator code to be replaced by modified version. If the code is to be used with newer versions of the TinyOS these modifications have to be migrated to new simulator code without breaking the structure of the new simulator. By commenting HAVE_TOSSIM_EXTENSIONS, SeMA sensor network application can be modified to work as an ordinary TinyOS application and it can be used with newer versions without any customization as a result sacrificing capabilities of the TOSSIM extensions. SEMATERMINATE definition is included in the hardware configuration if the application is required to terminate after the query timeout timer is reached. This timer is expected to reach on all nodes prior to termination so that any task that is in the progress of working is not interrupted. If this definition is commented, application will return to original state so that another query can be injected to the network.

The component graph of the SeMA sensor protocol for Mica2 hardware is given in Figure 5.19. The interfaces that are used by smRouterM module are wired to implementations in "smRouter.nc" file and this graph shows the resulting wiring. SmRouterM module's photo, temp and voltage interfaces are wired to Photo, Voltage and Temp modules so that whenever a getData command of ADC interface for photo interface is executed from smRouterM module, actual Photo module will be accessed. CC1000ControlM module is used changing the state of the radio interface of the sensor node. LedsC module is accessed when smRouterM application tries to activate green, red or yellow lamps on the node.

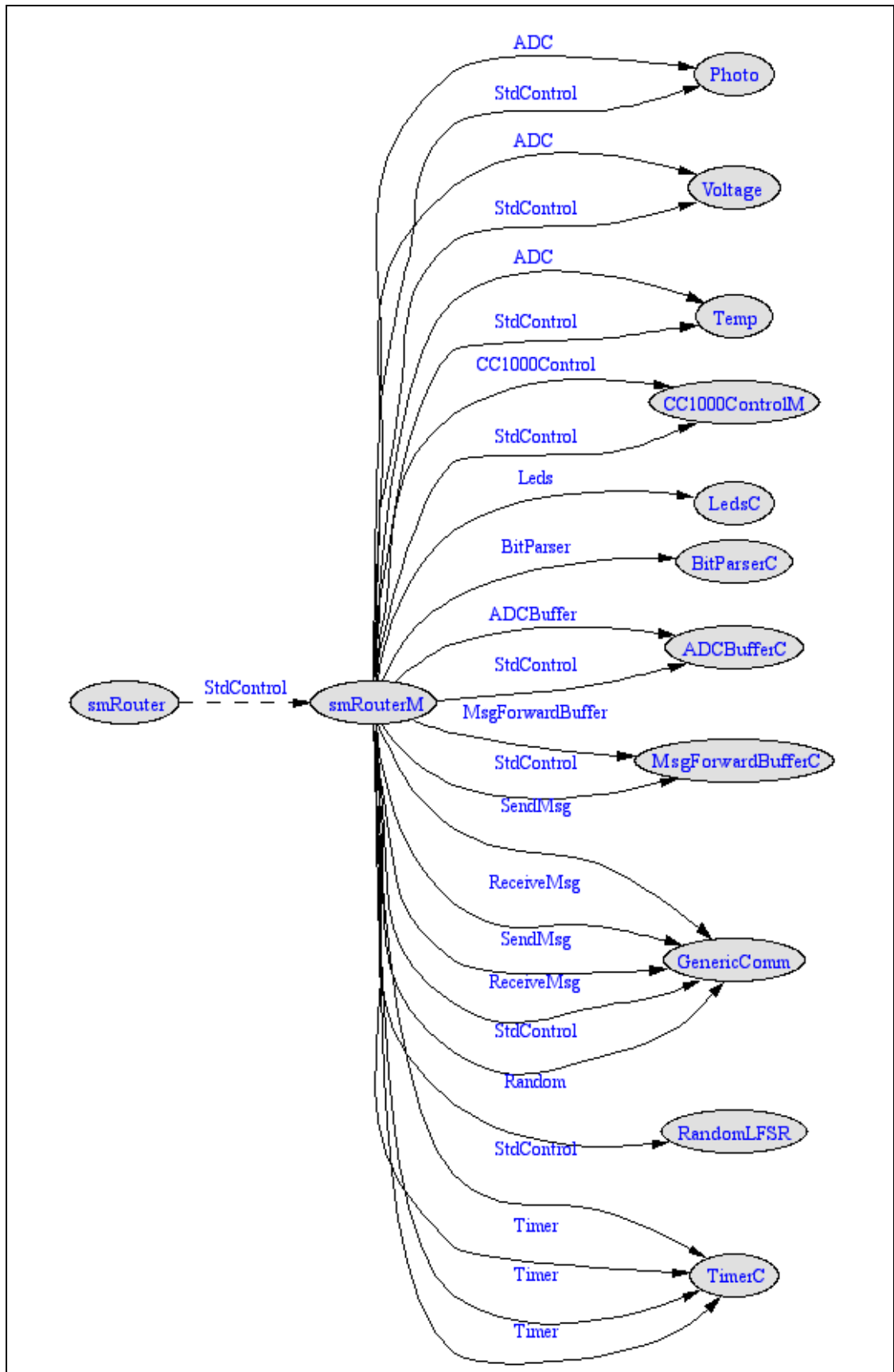


Figure 5.19 SeMA Querying Application Component Graph

Send and Receive calls are wired to GenericComm. SmRouterM module uses two different active message numbers for transmission and reception of the protocol messages: messages with AM ID of 20 are used for exchanging protocol packets and 30 is used by MoTeS for network monitoring. Three different timers, sampling period, query timeout and concatenation timer, in the system are wired to TimerC module so that their fired function is called when the timers are started and a timeout is occurred.

Besides from using existing modules provided by the operating system, three different modules were defined and implemented. These modules are:

BitParserC: Bitparser is a component that provides get and set commands to the users. The interface of BitParser is given in Figure 5.20. Get command is used to retrieve a specified sizeof value from specified position of the user provided buffer. Address of a 16 bit target variable is casted to 8 bit array pointer during function call. The variable provided to the function is used as a two sized 8 bit array in the module. Similarly, Set command is used to set a specified sizeof value at the specified position to the user provided buffer.

```
interface BitParser {
    command result_t Get(uint8_t *data,uint8_t pos,uint8_t size,uint8_t *target);
    command result_t Set(uint8_t *data,uint8_t pos,uint8_t size,uint16_t value);
}
```

Figure 5.20 BitParser Interface

MsgForwardBufferC: MsgForwardBufferC module is used for two different purposes by the application. First of all, it provides necessary abstraction between the MAC layer and the application. All received messages are stored in the buffers provided by this component instead of using MAC layer provided messages. Experience with coding proved that some MAC layers in simulation (packet level) does not handle buffer exchange correctly and results in improper corruption of the packet by other nodes while it is expected to be delivered to a target node. Therefore, all received buffers by receive functions are returned to MAC layer and their contents are copied to new buffers. Although this operation is costly, it avoids MAC layer problems from interfering the working of the protocol and also it provides application to keep a message as long as it wants, since the application owns the message.

```

Interface MsgForwardBuffer {

    command result_t Empty();

    command result_t Get(TOS_Msg **retmsg);

    command result_t Put(TOS_Msg *msg);

    command result_t GetEmptyBuffer(TOS_Msg **msg,u16 t);

    command result_t ReleaseEmptyBuffer(TOS_Msg *msg);

    command result_t GetTime(TOS_Msg *msg,u16 *t);

}

```

Figure 5.21 MsgForwardBuffer Interface

MsgForwardBufferC module provides the interface given in Figure 5.21. MsgForwardBufferC module contains two different data structures inside: empty message buffer store and incoming message queue.

As stated before, all received packets are stored locally. Application uses *GetEmptyBuffer* command to retrieve a buffer from the store and uses this message in order to perform parsing and processing of the message later at a task. When the application is done with the message, it uses *ReleaseEmptyBuffer* command to release the message to the store. For example, when a message transmission is requested by the application, application calls *GetEmptyBuffer* command to receive an empty buffer and fills the packet and lastly calls send command of the Send interface. The buffer is released when the application receives message transmission complete or failure notification from the MAC layer.

Besides from functioning as a node that resolves a query, nodes also act as a gateway between other nodes up to the sink node. Incoming message queue is used to store the received messages that are to be transmitted after being processed by the node. Calls to the *Send* command of the SendMsg interface for packet transmission can sometimes return false. Current RF hardware and MAC layer does not allow packet reception and transmission to occur at the same time. A message reception process can be in due while a node tries to transmit a message. In this case, the message is stored in the incoming message queue for retransmission at a later time and a task named *DeliverDataMessage* is posted.

ADCBufferC: All the sensing interfaces in the mote is connected to the ADC channel. Whenever a mote requests information from a sensing interface, the channel is kept busy. Similar to the message transmission case, sometimes calls to *getData* command of the ADC interface can fail due to a data gathering process. ADCBuffer is used to store the request in a queue so that when *dataReady* notification is made by the operating system, those data gathering requests waiting in the queue are activated one at a time. The interface provided is given in Figure 5.22. ID is the number of the sensing interface that is waiting in the queue to be queried.

```

interface ADCBuffer {
    async command result_t Empty();
    async command result_t Get(u16 *id);
    async command result_t Put(u16 id);
}

```

Figure 5.22 ADCBuffer Interface

The system functions as follows: The module is initialized with execution of RouterControl interface's init function. In this function, local variables are set to initial values by a call to *Initialize_LocalParams* function. Later all the components that are wired to the actual modules are initialized by executing each interface's init function call. Afterwards, start command of RouterControl interface is executed by the operating system and all sensing interfaces and communication interfaces are started and the system is set to IDLE state. Similarly, stop command of RouterControl interface is executed when sensor node decides to switch to power consumption mode.

In IDLE case, the system is expecting a setup message to start a query processing on the node. When a setup message is received, the system switches to ACTIVE state and incoming message is decoded and query parameters are stored to *subqueries* variable. Additionally, *sample_count_store* is used to make a backup of the actual sample counts in the setup packet. During data collection sample count is decremented and query processing is assumed to finish when sample count of all sub queries reach to zero. If the query is continuous, sample counts are restored to original values and another query processing is scheduled.

If the query includes a sensing related sub query, requests to sensing interface are made by *getData* call. *DataReady* event function of each sensing interface is executed when a data is fetched from the sensing interface. At each *dataReady*

function, *do_compilation* function is executed to store the sampled value to relevant positions of the *compilation_store* array. *Compilation_store* is used to store intermediary data during local processing. Compilation functions are applied to values stored in this array.

When a message is received from the radio interface, it is multiplexed to appropriate function by checking the message type and each message type is processed at relevant functions.

After completion of query processing, system initializes all local variables and transitions to *IDLE_STATE* again.

5.2.2 Driver Application

Driver is a java application that extends TinyOS provided java classes to allow parsing and creating of TinyOS messages that are coded according to AM message format. This application has been initially created using MIG application for listening to 29 bytes messages with AM ID of 20. This application also creates a TCP socket server listening to port 9998. If a message is received from the sensor network, *SerialForwarder* forwards the message to the Driver application and Driver application's *MessageReceived* function is executed. In this function, it's checked if there exists any application that is connected to the TCP port and received message is sent to the listening application from this socket. To allow two way communication between the driver application and the application that is connected to the socket, a thread that tries to receive message from the socket is created during startup. When a message is received from the socket, it is converted to TinyOS compatible message and transmitted to *SerialForwarder* using TinyOS provided java classes.

Having driver application that acts as a proxy in software architecture might seem as an overhead but it provides an abstraction from the TinyOS operating system. Any changes in packet reception routines of TinyOS would not effect SQS applications since inputs and outputs of SQS applications remain same.

5.2.3 Querier Application

Querier application is provided as an alternative means for retrieving XML service request attributes from SeMA architecture. Since the connection between SeMA architecture and sensor network has not been currently completed, this application is used to inject query parameters defined in the setup message.

User selects query related parameters from the GUI and injects the message using Send Setup Message button in the application which is shown in Figure 5.23.

The screenshot shows the 'sema_querier' application window. It contains several input fields and checkboxes for configuring queries. The main configuration area includes:

- Region ID:** 1
- Flooding Degree:** 20
- Hop Count:** 0
- Query Timeout:** 60 Minutes
- Acknowledge:** ☐
- Concatenate:** ☐
- Continuous:** ☒
- Query Frequency:** 5 Seconds

 Below these are four subquery configurations:

- Subquery 1:** Sensing Related, Sensor Type: Temperature, Sample Count: 5, Compilation Funct.: AVERAGE, Compilation Data: 0.
- Subquery 2:** Node Related, Sensor Type: Node ID, Sample Count: 1, Compilation Funct.: N/A, Compilation Data: 0.
- Subquery 3:** Node Related, Sensor Type: Parent ID, Sample Count: 1, Compilation Funct.: N/A, Compilation Data: 0.
- Subquery 4:** Sensing Related, Sensor Type: Photo, Sample Count: 3, Compilation Funct.: MAX, Compilation Data: 0.

 At the bottom, there is a 'Received Packets' listbox and an 'Adres' field with the value '127.0.0.1'. Two buttons, 'Clear' and 'Send Setup Message', are located at the bottom right.

Figure 5.23 Querier Application

When user issues send command, selected options are converted to SeMA bitwise encoded packets and then querier application connects to the driver application using TCP sockets and sends the message to the driver application in SeMA message format which constitutes the 29 bytes data section of TinyOS message. The node which is connected to the programmer board transmits setup message to the network and responses are delivered to the querier application following SerialForwarder and driver path. The results retrieved from the sensor network are decoded from the packet and displayed in the listbox found in Received Packets sections of the GUI. In order to keep the consistency, the message format defined in this application has to be kept same with the TinyOS message format. Otherwise, messages received would be decoded incorrectly.

5.2.4 Oscilloscope Application

Oscilloscope is an application that was developed by University of California, Berkeley and it has been released to the public use by TinyOS operating system. Oscilloscope application uses its own messaging mechanism to draw the change of

the values over time for some duration. Experiences with TinyOS operating system have proved that Oscilloscope application is a vital part of sensor network application tool suite. In order to use this application's capability, protocol conversion has been done. Data retrieved from sensor network in SeMA encoded packets are encapsulated in Oscilloscope application's packet format and oscilloscope application messages have been injected to java oscilloscope application using TCP sockets. Conversion is done in querier application. Querier application instantiates a TCP server that listens to socket 8000 and when oscilloscope application is started it connects to the querier application and fetches the values from querier and injects the message to Oscilloscope GUI classes' methods.

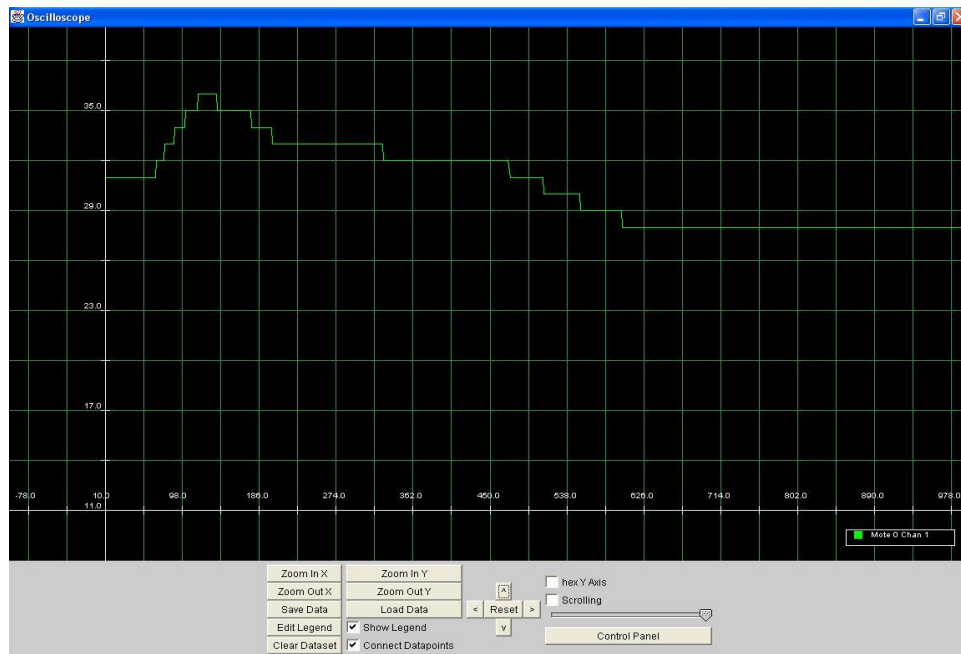


Figure 5.24 Oscilloscope Application

The GUI of oscilloscope application is shown in Figure 5.24, the plot includes temperature change of the room monitored by sensor motes. Oscilloscope application uses different channels for monitoring different data as in the case of a real oscilloscope. Channel information has been converted from querier application in the form of hop count so that data received from different hops can be monitored using this application.

5.2.5 External Data Binding Mechanisms

Sensor network part of the architecture is as stated before independent of SeMA architecture and can be used for other purposes. In order to achieve this flexibility,

several proxy applications such as querier and driver have been written to abstract details of each layer from interfering each other. However, having these applications don't make life still more comfortable. It is still required that received data messages have to be processed to decode sensor data. To extend the usage of the architecture, data received by querier application is dumped to three different tables in a database so that any user who doesn't have any idea about sensor networks can write software that accesses database using Open Database Connectivity (ODBC) connection and further use the results for his own use. Sensor network becomes a database application for this user.

MySQL server has been chosen as a storage server and MySQL ODBC client has been used to access MySQL server as an ordinary database server. Any database server can be used provided that a Data Source Name (DSN) named sema is created in ODBC data sources.

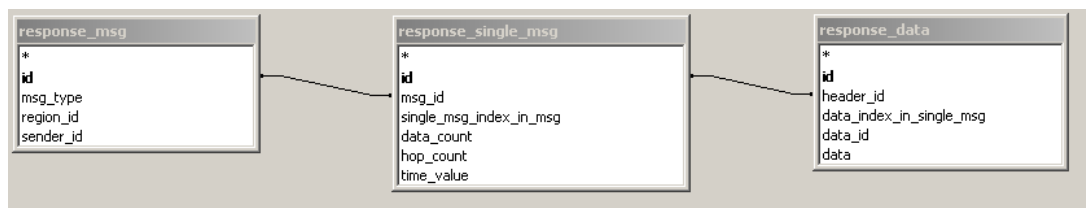


Figure 5.25 Database Design

Three tables named response_msg, response_single_msg and response_data were created in the database. The relationship between these tables is given in Figure 5.25. The protocol discussed in Chapter 4 for response delivery has been taken into account for the design. After the database has been created, any ODBC aware application can be used to access stored data. A sample snapshot of the data retrieved from the sensor network is given in Figure 5.26.

	id	msg_type	region_id	sender_id	single_msg_index_in_msg	data_count	hop_count	time_value	data_id
▶	641	1	1	65535	0	2	1	15012	0
	641	1	1	65535	0	2	1	15012	1
	641	1	1	65535	1	2	1	15012	0
	641	1	1	65535	1	2	1	15012	1
	641	1	1	65535	2	2	1	15012	0
	641	1	1	65535	2	2	1	15012	1

Figure 5.26 ODBC Access of Sensor Data from The Database

6 EXPERIMENTS AND TESTING

Sensor node software has been developed using TinyOS operating system and NesC programming language over MICA2 & MICA2Dot hardware [32]. TinyOS simulator, TOSSIM has been used to test algorithms for scalability. As mentioned before, TOSSIM is a discrete event simulator for TinyOS. It allows code written for TinyOS to be used for simulation directly. Primary goal of TOSSIM simulator is to emulate how TinyOS works on motes by using the same components of TinyOS and changing behavior of system dependent code. For example TOSSIM interrupts do not preempt each other and tasks are not interrupted by TOSSIM interrupts either.

TOSSIM has been used extensively to test the algorithms against certain inputs. Debug messages have been used to trace how applications perform. Custom developed applications such as message injector that use TOSSIM sockets to inject setup messages and collect response messages have also been developed.

6.1 Packet Level Data Transmission Add-on for TOSSIM

TOSSIM uses ADC and RFM models to simulate different communication scenarios and different ADC readings that can be achieved. Lossy model models node connectivity with graph edges and loss rates for that link. A customized version of lossy model has been used for our tests. According to [34], TOSSIM uses mica communication stack at bit level simulation so that MAC protocols also effect how messages are transmitted. However, our tests proved that simulation of radio communication is unstable and results in higher rates of messages getting lost as well as some messages not being delivered because of inaccurately designed TinyOS Active Message Acknowledgement methodology [34]. Additionally, Mica2 stack and other MAC protocols such as SMAC [40] are proved to be much stable than MICA stack and we didn't want to base our responses to a specific kind of MAC protocol since proposed architecture design is independent of the underlying MAC Protocol. As a result, simulator's bit level communication stack has been replaced with packet level communication stack by replacing the bindings of packet

transmission component wiring in “RadioCrcPacket.nc” file to a newly developed packet transmission module named as “RadioCRCPacketM.nc”.

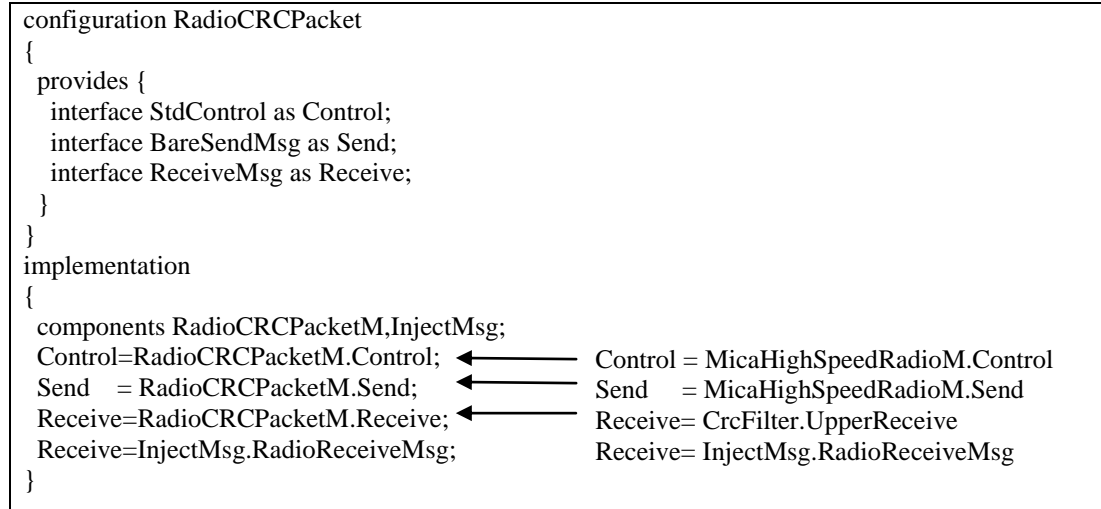


Figure 6.1 Packet Level Transmission Configuration

TOSSIM simulates the behavior of Mica MAC stack with MicaHighSpeedRadio module and in Figure 6.1 how packet transmission, reception and control interfaces of the simulation are wired to the RadioCRCPacketM module’s interfaces is shown. After applying this configuration, when a packet is received by simulation it’s fed to RadioCRCPacketM’s receive function. Similarly each transmit request by the application is fed to RadioCRCPacketM’s Send function.

$$P(NoBER) = 1 - P(BER) \quad (6.1)$$

Packet delivery has been utilized with bit error rates from lossy model and probabilistic collision using Equations 6.1, 6.1, 6.2, 6.3, 6.4, 6.5 and 6.6. Since Equation 6.2 contains packet length, packet loss probability is different for various sizes of packets.

$$P(NoPER) = P(NoBER)^{packet_length} \quad (6.2)$$

However, not all of the packet errors are caused by bit error rates. Collisions occur when nodes try to send data to a clear channel at the same time. CSMA allows nodes to detect if another node is already transmitting; however, proposed MAC protocols do not include carrier detection. Therefore, two nodes can see the medium free and transmit at the same time.

$$P(Col) = \sum_{i=2}^n C_N^i \cdot \alpha^i \cdot (1-\alpha)^{(N-i)} \quad (6.3)$$

According to Equation 6.3, collision probability is the sum of probability of two or more nodes transmitting at the same time.

$$P(NoCol) = 1 - P(Col) \quad (6.4)$$

Losses caused by the MAC layer have been modeled as in Equation 6.3. In the equation, N is the number of neighboring nodes in the same transmission range with transmitting node and α is the probability that at least two nodes will transmit a packet at the same time after detecting the channel as clear.

$$P(tx_success) = P(NoBER) \cdot P(NoCol) \quad (6.5)$$

According to these equations, for each packet to be transmitted, a packet error rate (PER) is computed using given link bit error rates by lossy model input file as in Figure 5.15. After that, the probability of packet's being lost in collision is computed using Equation 6.3.

$$P(PLP) = 1 - P(tx_success) \quad (6.6)$$

6.2 Power Management Add-on for TOSSIM

Currently, there's no power management component in TOSSIM. Therefore, a simple battery simulation environment for the simulator has been established for tests. After the simulation is started, each node is assigned a battery level parameter as 3000 mAh which is taken from a standard battery. At each packet transmission and reception, this value is decremented by transmission and reception powers taken from [41] respectively. Sleep energy of 30uA taken from [42] is decremented between each successive simulation events. During packet transmission, the battery level of the node that data is to be transmitted to is checked and if it's below 0, no packet is transmitted to that node which means that node has run out of power.

6.3 Performance and Functionality Tests

Prior to starting simulation with TOSSIM, we have created simulation parameters offline and we inspected the results from TOSSIM after running the simulation. A trace generator tool has been used to create events and position nodes in the terrain.

Table 6.1 Simulation Parameters

Environment Constants	Values
Terrain Size	200m X 200m
Transmission Range	30m
Sensing Range	30m
Simulation Time	300 simulation seconds
α	0.05
Bit Error Rate (BER)	0.00001
Number Of Events Generated	80
Number Of Nodes Used	20,40,60,80,100
Number Of Generated Topologies per node number	10
Number Of Tests repeated for each topology	30
Event Duration	10 - 15 Seconds

Parameters used in the simulation can be seen on Table 6.1. A terrain size of 200 meters x 200 meters and a transmission range and a sensing range of 30 meters have been used. In order to use results from these output files, a new ADC model has also been written. This ADC model reads the output files and fills the necessary data structures in the simulator for node positions and lossy model parameters.

Event generator tool generates events in the terrain to last for some duration at random positions and startup times. When the simulation is started with the SeMA ADC model, simulator checks for each node if the event is in the sensing range of the node and creates a simulation event to take place at event start time away from setup message reception time and removes the event after event duration time has passed. During this time, if a node tries to gather data from ADC it detects the event.

Used query parameters for simulation can be seen on Table 6.2. A continuous query message with 300 seconds QTP, 5 seconds S has been used. Each node tries to read a value lower than 65535 every 5 seconds and when new ADC model is used, it returns 1 if it detects an event registered for the node; otherwise, it returns 65535. So a message will be transmitted by the node if it detects an event.

Table 6.2 Query Parameters Used For Simulation

Query Parameter	Values
Flooding Degree	20
Region ID	1
Continuous Flag	True
Query Timeout	300 seconds
Sampling Period	5 seconds
Subquery Count	1
Sensor Type	Temperature
Sample Count	1
Compilation Function	Lower
Compilation Data	65535

Tests for 20, 40, 60, 80 and 100 nodes have been made by injecting the same query for all tests to see how well architecture performs. Additionally, 10 different topologies, that is tested for different number of nodes, have been generated and simulation tests for aggregate and immediate way of delivery for each topology and node count have been run so that sensor node positions and terrain have been kept same while inspecting the effects of aggregation over query processing. Moreover, 30 tests for each topology have been made and the average of the results has been taken in order to reach to a result distribution and remove unnecessary effects over total system behavior.

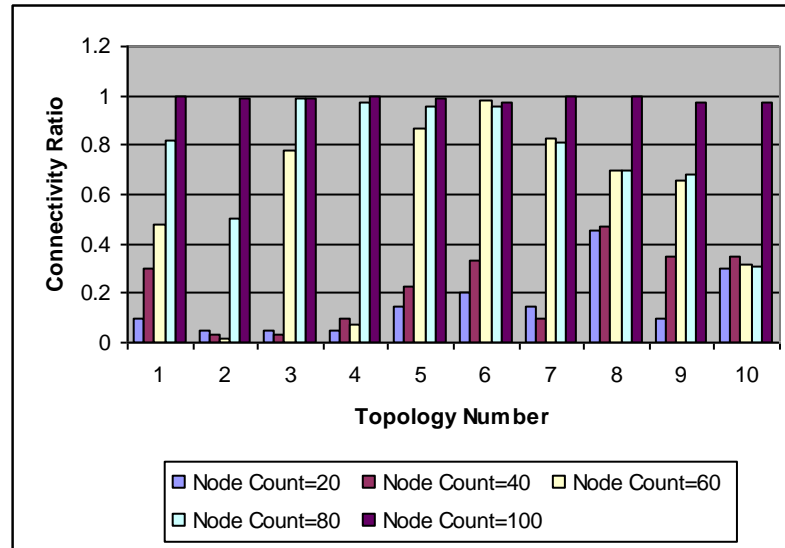


Figure 6.2 Connectivity for Immediate Delivery

With this experiment setup, connectivity rate, region coverage, event delivery rate and message delivery rate for different number of nodes and different delivery mechanisms have been measured. Node connectivity distribution can be seen in Figure 6.2 for immediate and aggregate event delivery. Since connectivity is a

function of the topology and setup message distribution connectivity results have been similar for aggregate and immediate event delivery.

Connectivity Ratio is the ratio of the number of nodes joining query processing to the total number of nodes in the terrain. From Figure 6.2 it can be seen that higher number of nodes result in higher rates of connectivity in both graphs since the node density in each region becomes higher as node count increases, the number of nodes receiving setup packet also increase. For some test topologies connectivity ratio seems to decrease while increasing node count for some time and then a tremendous increase in connectivity is established. This is due to the far node placement in the terrain. As node number in the terrain increases, nodes will be able send packets to each other and from some point all of the nodes will be accessed. This problem is named as network partitioning in the literature.

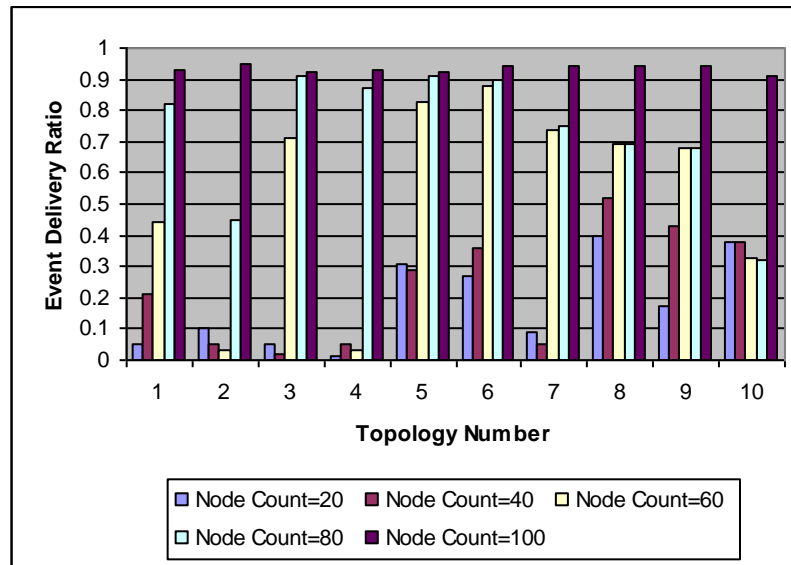


Figure 6.3 Immediate Event Delivery

Event Delivery Ratio is the ratio of number of events delivered to monitoring application to total number of events registered for all nodes including nodes that are not connected in the terrain to be detected. If the links were lossless and all nodes in the terrain were connected to each other, event delivery ratio would be expected to be 100%.

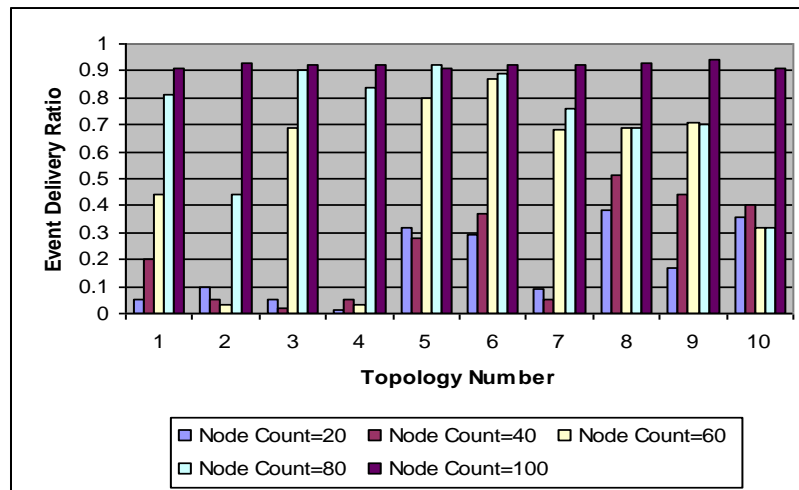


Figure 6.4 Event Delivery for Aggregate Event Delivery

It can be seen from Figure 6.3 and Figure 6.4 that as the number of nodes increase, the number of events delivered to root node increases as well. Thus, more nodes will be connected and received events will increase as well. In the case of lower node density, events will be registered for nodes to be detected but as some nodes might not receive setup message in the case of a far distribution between the nodes. As a result, some of the events would be missed. Similar results have been achieved for both aggregate and immediate event delivery. Nevertheless, because of the reason that in aggregate event delivery nodes carry one message to deliver more than one event message, when a message is lost in aggregated event delivery more events won't be able to be delivered when classified with immediate delivery. That's why; differences between these two graphs exist.

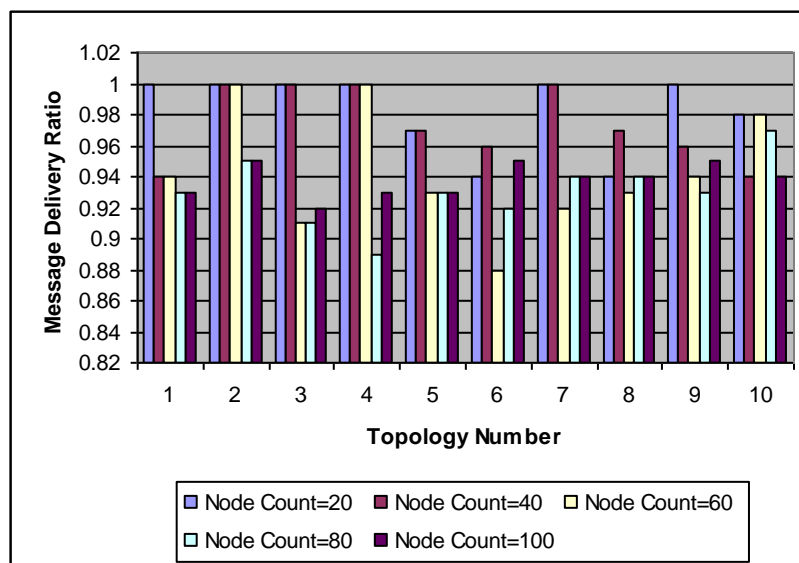


Figure 6.5 Message Delivery Ratio for Immediate Delivery

Message Delivery Ratio is the ratio of messages delivered to the root to the total number of events detected. If immediate mode of delivery is selected and the links are lossless, Message Delivery Ratio will be expected to be 1. It can be inferred from Figure 6.5 that as the number of nodes increase, the number of messages successfully delivered decrease. Since node density increase with increasing node number, the probability of messages not being delivered will increase. Figure 6.5 shows the effect of packet losses in the simulation. Packet Loss Probability (PLP) of 0.10 has been experienced with 100 nodes during simulation.

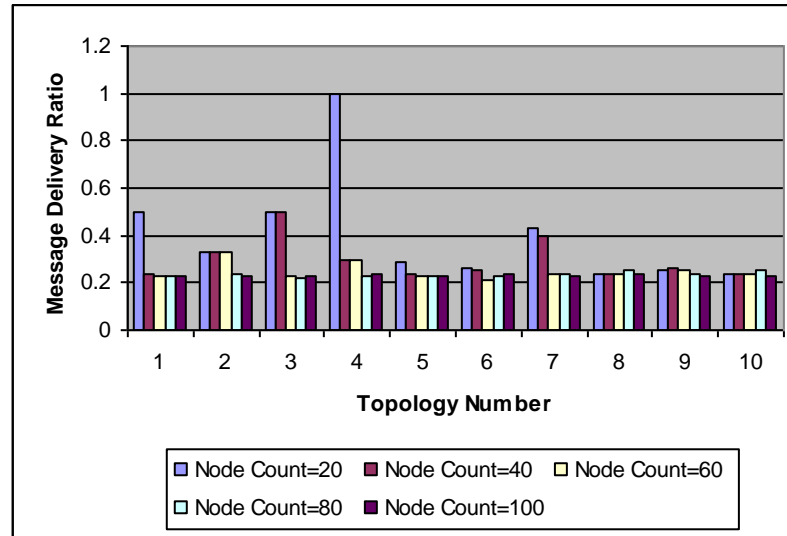


Figure 6.6 Message Delivery Ratio for Aggregate Message Delivery

When Figure 6.6 is inspected, Message Delivery Ratio seems to be quite low. However, lower message delivery ratio means lower number of messages being transmitted in the terrain to detect the events. Since response messages are concatenated, one message carries at a maximum of four response messages. Therefore, event delivery rates of 80% has been achieved with 100 nodes by only transmitting 20% of total messages while similar results with immediate delivery can only be achieved with transmitting 78% of all messages.

Figure 6.6 also shows the change trend of message delivery versus node count. For aggregated messages as the number of nodes increases it eventually reaches to 0.2 message delivery since in that case all nodes that want to transmit message can find a neighboring node's message to aggregate. This indicates that concatenation is applicable to dense sensor networks.

6.4 Timing Tests

Another set of experiments have been carried out using simulator to inspect the message delivery characteristics of sensor nodes. Simulation parameters that were used are shown in Table 6.3. A continuous query with QTP 60 seconds and S 5 seconds has been used for temperature sensor to collect data reading every 5 seconds without applying any compilation functions.

Table 6.3 Timing Test Simulation Parameters

Query Parameter	Values
Region ID	1
Continuous Flag	True
Query Timeout	60 seconds
Sampling Period	5 seconds
Subquery Count	1
Sensor Type	Temperature
Sample Count	1
Compilation Function	0 (N/A)

As a result, 12 messages per node are expected to reach driver node when immediate message delivery is used. Flooding degree (fd) parameter has been varied between 1 and 10 during this experiment. Variation of fd results in different number of nodes to be connected. It is to be analyzed that with which fd parameter can nodes cover the region and how long driver node would wait incase it uses aggregate message delivery.

In this experiment, a static sensor configuration for node positions which was generated by topology generator tool has been used. Additionally, lossy RFM model has been used to simulate link losses with previously described PLP calculation formulas. SeMA ADC model has been left out for this experiment and standard ADC model that generates random number is used for the reason that we're not interested in any events that is to be detected. A terrain size of 200m x200m was used for simulation with 100 nodes, 30m transmission range and 30m sensing range. The node distribution was randomly generated and it is shown in Figure 6.7.

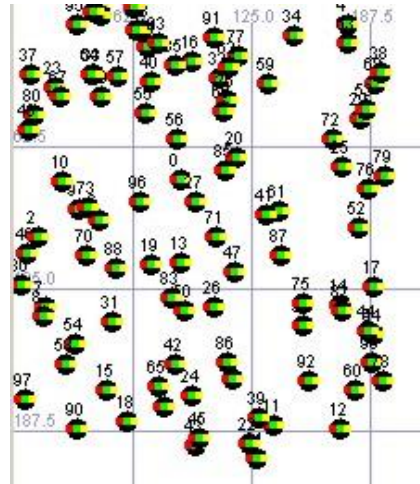


Figure 6.7 Timing Test Topology

In this experiment, previously discussed timing mechanism has been used with a slight change. When each node receives setup message, it sets up a counter which is incremented by each S timer tick and this counter value is used when a message is allocated and to be transmitted to find time difference. Therefore, the precision of the elapsed time depends on the time unit of the S timer. More precise timers could be employed, but when sampling time is set to seconds or minutes, calculated time in the mote remains negligible among others. In fact, it has been measured that the average time for a mote to transmit a message after using it is in terms of microseconds. Data throughput of 12kbps has been used with message length to calculate the message propagation time and it has been added to each transmitted message.

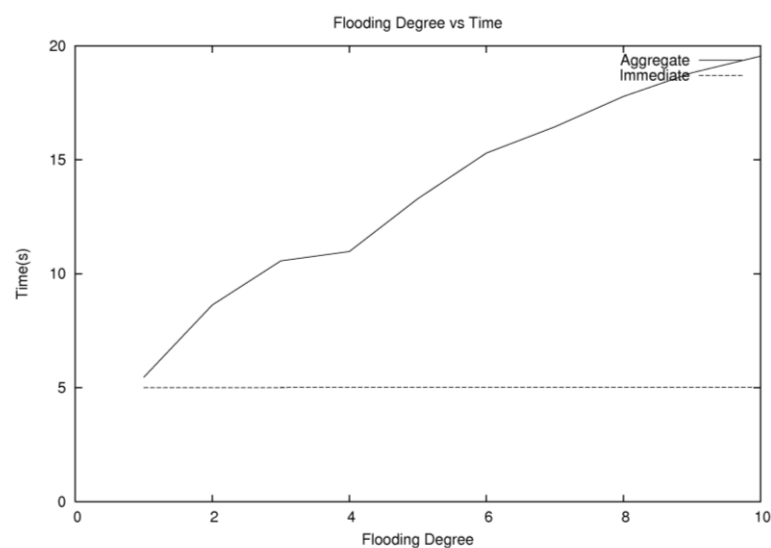


Figure 6.8 Flooding Degree vs. Average Response Time

fd has been varied between 1 and 10 and it has been incremented by one after making simulation tests for 20 times at each fd value. During the simulation, elapsed time of each received message at root node (node 0) has been used to calculate average message reception time and average of each calculated average message reception time after 20 tests has been used for plots in Figure 6.8 and Figure 6.9.

It can be seen in Figure 6.8 that when aggregate mode of message delivery is used the message reception time increases while message reception time for immediate message delivery remains nearly same. Delayed message reception is caused by concatenation timer that is set at each node. According to Equation 4.5 when the flooding degree parameter increases the time a node waits for other nodes' messages to transmit its own message increases. However, as fd parameter increases the area coverage increases. Therefore, the time it takes a message to be filled decreases. It is the reason why the curve advances with a more slope than a directional relationship with flooding degree.

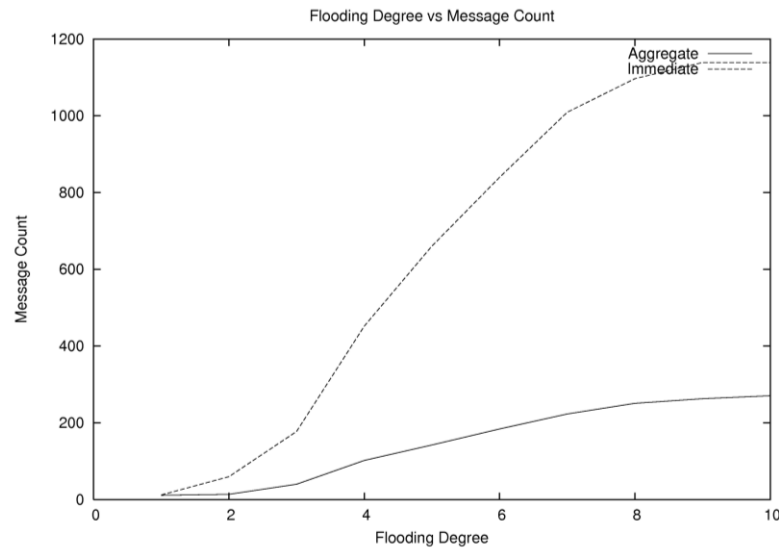


Figure 6.9 Flooding Degree vs. Message Count

Figure 6.9 illustrates that as fd increases the number of received message increases as well, this is because of the fact that the number of nodes joining query processing increases. However, there is a tremendous difference between aggregate and immediate mode of message delivery. As there is 100 nodes in the terrain and it's expected that each node send 12 message per simulation it can be seen that with fd of 9 whole area is covered since it no longer increases and message reception rate is very close to 1200. The difference between these values is caused by MAC layer

collisions and link layer losses. As in the previous experiment, a message delivery proportion of 1/4 has been achieved with this test as well.

6.5 Data Traffic Analysis

Local processing is one of the key features of SQS query processing system. Use of SC allows nodes to perform local data compilation prior to transmission. Transmission is the most costly operation in sensor networks and by the use of distributed local data processing on every node tremendous message count decrease is achieved.

In order to see the effects of local processing prior to data transmission simple mathematical equations applying to general in-network processing communication schemes is used to find difference. In order to generalize the mathematical approach, it is assumed that every node transmits a data message and a negotiation message to elect cluster leaders or parent node on the tree. The worst and best cases are handled separately. The best case is that every node is connected to a central base station and each transmitted message is converted to a total of one processed message according to in-network processing. The worst case is that every node is connected to exactly one node creating a chain with maximum hop count and transmission count to drain energy. These cases are seen in Figure 6.10.

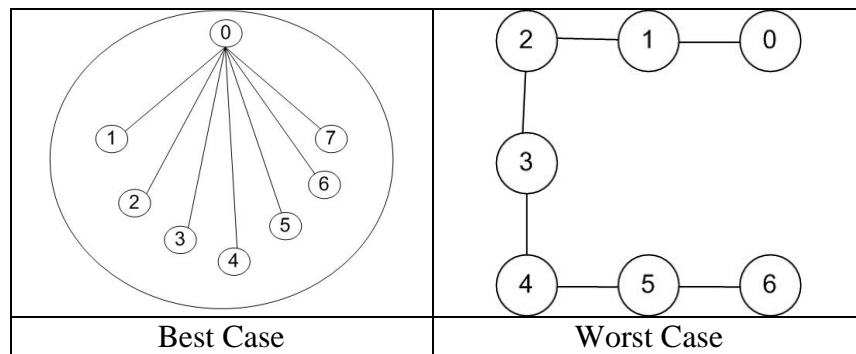


Figure 6.10 The Best and Worst Cases

Let N be the total number of nodes in a terrain, S be the sampling period for an in-network processing scheme, QTP be the total time this query is expected to run and SC be the sample count of the SQS query system. Every node relays a message every QTP/S or $QCycle$ time. If the message came in a way to achieve aggregation and compacting the received whole packet, best case situation occurs and node returns the minimum number of messages. If no messages have been received to calculate

aggregated result then the maximum result is reached with the worst case. As a result total number of messages transmitted in a network would be in the range as shown in equation 6.7.

$$(\frac{QTP}{S} + 1).N < message_count < (N + \frac{QTP}{S} \cdot \frac{N.(N+1)}{2}) \quad (6.7)$$

If local processing is employed in this system, instead of transmitting message every QTP/S each node will collect SC times data from the ADC and apply aggregation function prior to transmission and relay only one message per node. If each parent node or cluster head also applies aggregation functions, the number of messages in the network will decrease tremendously. SC can be chosen by the user, or can be varied dynamically or can be a constant. For example if a user requested message every 1 second, SC can be chosen 10 and the result will be delivered to sink node at the 10th second.

$$(\frac{QTP}{S.SC} + 1).N < message_count < (N + \frac{QTP}{S.SC} \cdot \frac{N.(N+1)}{2}) \quad (6.8)$$

If SC is instrumented into the network, message count in the network will change as in equation 6.8. This equation states that if RGP (QTP/S) remains big enough when compared with N then local aggregation will decrease the maximum and minimum number of messages by a multiple of SC.

The same topology in Figure 6.7 and simulation environment used in timing tests have been used for data traffic analysis of TinyDB query processing system and SQS system with immediate and aggregate delivery. Query parameters used in the simulation are in shown in Table 6.4 and the connectivity ratios of the nodes for different number of nodes are given in Table 6.5.

Table 6.4 Local Processing Test Query Parameters

Query Parameters	Values
Flooding Degree	20
Query Type	Continuous
Sampling Period	1 Seconds
Sample Count	1,5
Sensor Type	Temperature sensor
Compilation Function	Average

Four different cases have been created. The aim of each of these cases is to compare the effect of the usage of different processing techniques to the network utilization. The terrain has been queried for 30 seconds to retrieve the average temperature in all of the cases. The results are shown in Figure 6.12. The first case labeled as “SQS Immediate Delivery SC=1” in the graph tries to collect average temperature every second using SQS. Since SQS doesn’t perform in-network processing this query is typically equal to “retrieve temperature every second”. The second case, labeled as “SQS Immediate Delivery SC=5” performs local processing. The third case, labeled as “TinyDB in-network processing” collects data from sensor nodes every 1 second and results are processed using in-network processing. The last case, labeled as "SQS aggregate delivery SC=5" applies concatenation to resulting packets to further decrease the number of transmitted messages.

Table 6.5 Connectivity Ratio for Data Traffic Simulation

Node Count	Connectivity Ratio
30	0.2
50	0.56
70	1
100	1

Network utilization of the first case is the worst/highest as expected since no processing is done to the collected messages. Every node relays 30 messages during this time and as a result of multi hop communication messages are retransmitted to deliver each packet to the root node. Therefore, greater number of messages transmitted by each node results in more retransmission and worse utilization.

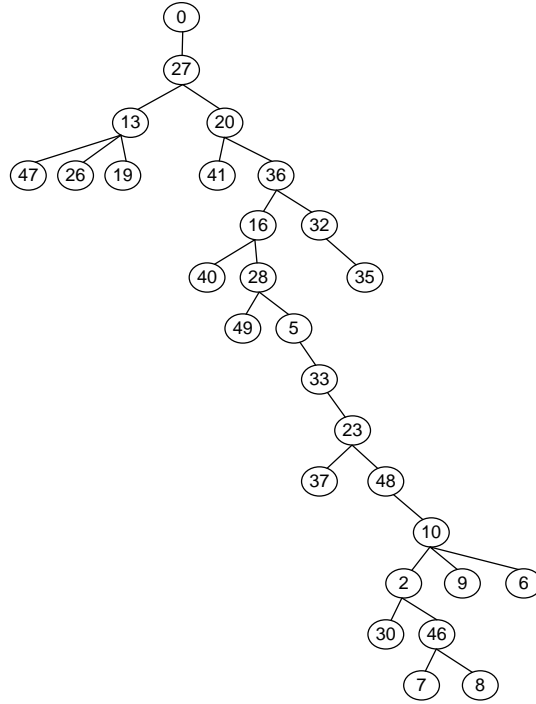


Figure 6.11 Logical Topology for Data Traffic Simulation

To see the effect of local processing SQS system has been run in simulation with a SC of 5 so that a message is generated every 5 seconds and the average of temperature is calculated during this time for 5 samples. Simulation has been terminated after 30 seconds again. Theoretically, 6 messages per node are expected from SQS system if all of the links are lossless and no hidden terminal problem or other MAC layer problem is encountered. However for 50 nodes with the logical graph in Figure 6.11, 153 messages have been delivered to the root node instead of 168 messages resulting in 10% loss by application of Equation 6.6 to the transmitted messages. As a result of multi hop communication, total number of messages transmitted in the network is 1035.

TinyDB has been run on simulation environment with an SQL expression of "*select avg(temp) epoch duration 1024* " to collect temperature values from nodes every second and simulation has been terminated after 30th message so that simulation time is limited by 30 seconds. TinyDB nodes transmit extra control messages to each other to elect processing leaders and data messages transmitted by the neighbors are processed using in-network aggregation and one message that contains processed result is relayed instead of transmitting each message which is the first case. The ratio of control messages to the transmitted data message has been 0.05% for 100 nodes. Therefore, in order to compare the results with SQS which does not contain

control messages, TinyDB results exclude the number of control messages which is negligible compared to the total number of data messages.

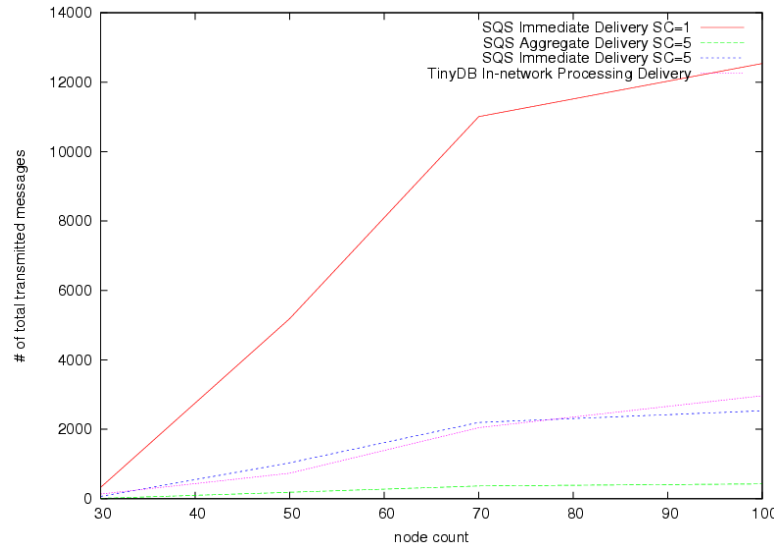


Figure 6.12 Data Traffic Analysis

The last case uses in-network message concatenation in addition to local processing wherever possible to decrease the number of total messages and transmit each packet at the full packet length.

When Figure 6.12 is inspected, it can be seen that first case performs worst and has message counts in the magnitudes of 10000. The difference between second case(local processing only) and first case, shows the effect of local processing to straight forward data transmission which is slightly bigger than a multiple of 5 as expected by the ratio of SC=1 to SC=5 and from the equations in evaluation section. The performance of in-network processing scheme of TinyDB (case 3) and local processing is competing in general. TinyDB performs better up to node count of 70, later on local processing becomes first. The forth case, SQS concatenation of locally processed messages performs best according to the simulation results. Since SQS concatenation can wrap up to four messages into one message, 4 times better results are achieved when compared to SQS locally processed simulation and TinyDB.

Simulation results prove that local processing is a critical design issue for sensor network databases since it can compete with in-network processing schemes. Generally, SC is dependent on the user's data request interval. Therefore, provided that user can specify data acquisition interval and duration, local processing schemes must be incorporated into query processing systems. It has been also seen that in-

network processing contributes well to the energy consumption. Consequently, a scalable query processing system for wireless micro sensors is expected to integrate in-network processing, local data processing and message concatenation data processing techniques.

7 CONCLUSION

In this thesis, a scalable system for topology construction and query processing mechanism for data collection among tiny sensors have been presented. The overall model is a three tier architecture constructed to process a query devised in an Internet application down at a group of sensors in the target domain. A MANET is assumed to carry XML formed queries and data between the end-user application and a group of sensor drivers. Each driver establishes a network for query processing on its dynamically associated sensors. The parameter driven scalable structure of the proposed model makes it usable for querying in general purpose environmental monitoring applications. Solutions for a number of problems are proposed: topology construction via query dissemination, response aggregation for energy optimization via in-network buffering and message concatenation, employing of local processing algorithms for distributed data processing prior to transmission and handling timeouts efficiently.

TinyDB, Cougar and Acquire use in network processing to decrease the number of messages. TinyDB inspects response characteristics of aggregation operators such as average, sum and count and defines methods to carry intermediary results. Acquire uses active queries to resolve queries partially at each step and each sensor node functions as a cluster head to resolve part of the query from a transmission diameter. Similarly, Cougar resolves user queries by generating query plans and assigning some nodes to be cluster heads to resolve part of the queries. SQS also make use of limited sensor processing power but instead of applying aggregation functions at the reception of messages that contain raw sensor readings on the intermediary nodes, SQS prefers taking a user defined sample count times sensor reading and apply user selected compilation function to the result and relay resulting data to sensor node's one hop away parent. Packet transmission is the most energy consuming operation and SQS tries to limit this operation. This kind of local processing has been named by TinyDB as temporal queries and is partially supported in two steps. First, TinyDB makes use of the flash memory to store sensor readings by issuing a sensor query to ask sensor nodes to put results into tables in the flash memory and then with a second

query sensor results are queried from these tables. This two step operation that requires user interaction puts unnecessary delay to user requested data. Additionally, usage of flash memory is also as energy consuming as packet transmission and is not desirable to use. This is the reason why flash memory logging is disabled by default in TinyDB. Packet overhead is also reduced in SQS by delaying route maintenance until data delivery, whereas in TinyDB, root node periodically announces itself in control packets so that sensors can update their parents and maintain the routes towards the root. For periodic queries, top-down tree maintenance with control packets can be justified whereas for queries returning rare events such as “return sensor readings if temperature $>50^{\circ}\text{C}$ for the next one hour”, overwhelming control packet processing may reduce the overall network lifetime as this query may never generate data traffic at all. Additionally, SQS returns timing information in each data packet to be used to predict events in relation to the depth of the return path so that nodes are not required to be time synchronized with each other which cause extra overhead.

There are also some differences in architectural assumptions. The upper tier of SeMA architecture supports XML format for service definition. Thus, underlying sensor network infrastructure can easily be made transparent to web-based monitoring applications conforming to XML standards.

SQS has been implemented on mica2 and mica2dot hardware running TinyOS operating system to determine the limitations and performance characteristics of the protocols. The architecture is analyzed for response generation characteristics of different query types. With the proposed energy saving optimization methods, it has been observed that the architecture is flexible enough to handle diverse environmental monitoring applications with different requirements as well as network monitoring for administrative purposes. As the protocols are implemented in TinyOS active messages payload, packets sizes are limited with 29 bytes. A number of limitations exist such as; maximum of four sub queries are supported because of packet size of TinyOS Active Message payload size. Maximum of four response packets can be concatenated again because of packet size limitations. Although a wide range of compilation functions are supported, they are currently statically built into the system software. Additionally, as Sampling Period parameter is globally set

for all sub queries in the current implementation, this can also be considered as a limitation.

REFERENCES

- [1] **Agrawal,D.P Zeng,Q.A.**,2003.Introduction to Wireless and Mobile Systems, Thomson Brooks/Cole, Pacific Grove, CA 93950,USA.
- [2] **Tanenbaum,A.S.**,1996.Computer Networks, Prentice Hall, Upper Saddle River, NJ, USA.
- [3] **Rubin,I Behzad,A Zhang,R Luo,H and Caballero,E.**,2002. TBONE: A mobile-backbone protocol for ad hoc wireless networks with unmanned vehicles, *IEEE Aerospace Conference Proceedings*, Big City MT ,USA ,March 2002,Volume: 6, 2727 -2740.
- [4] **Akyildiz, I. F. Su, W. Sankasubramaniam, Y. and Cayirci, E.** 2002.Wireless Sensor Networks: A Survey. *Computer Networks*, **38**:393–422
- [5] **Defense Advanced Research Projects Agency MEMS Program Overview and Vision**, <http://www.darpa.mil/mto/mems/>.
- [6] **Kahn, J.M. Katz, R.H. and Pister, K.S.J.**,1999. Next Century Challenges: Mobile Networking for Smart Dust, *ACM/IEEE Intl. Conf. on Mobile Computing and Networking*, Seattle, WA,USA, August 17-19, 271-278.
- [7] **Maurer,W.**,2003.The Scientist and Engineer’s Guide to TinyOS Programming ,DspLabs, Wedgewood Way Livermore ,CA 94550,USA.
- [8] **Rai, L. Kotapati,K.**,2003. A Survey on Sensor Network, 2003, http://www.cse.psu.edu/~cg530/proj03/Sensor_Networks.pdf.
- [9] **Ning,X.** A Survey of Sensor Network Applications, <http://enl.usc.edu/~ningxu/papers/survey.pdf> .

- [10] **Mani B. Srivastava, M.B. Muntz, R.R. and Potkonjak, M.**, 2001. Smart kindergarten: sensorbased wireless networks for smart developmental problem-solving environments, *Proceedings of the ACM SIGMOBILE 7th Annual International Conference on Mobile Computing and Networking*, Rome, Italy, July 2001, 132-138.
- [11] **Hedetniemi, S., Hedetniemi, S., and Liestman, A.**, 1988. A Survey of Gossiping and Broadcasting in Communication Networks. *Networks* **18**, 319-349, 1988.
- [12] **Weisstein, E.W. et al.**, Gossiping. From [MathWorld](http://mathworld.wolfram.com/Gossiping.html)--A Wolfram Web Resource. <http://mathworld.wolfram.com/Gossiping.html>
- [13] **Tijdeman, R.**, 1971. On a Telephone Problem. *Nieuw Archief voor Wiskunde* **19**, 188-192, 1971.
- [14] **Harary, F. and Schwenk, A. J.**, 1974. "The Communication Problem on Graphs and Digraphs." *J. Franklin Inst.*, **297**, 491-495, 1974.
- [15] **Kulik, J. Rabiner, W. and Balakrishnan. H.**, 1999. Adaptive protocols for information dissemination in wireless sensor networks. *Proceedings of the 5th Annual IEEE/Mobicom Conference*, Seattle, WA, August 1999, 174-185.
- [16] **Intanagonwiwat, C. Govindan, R. and Estrin, D.**, 2000. Directed diffusion: A scalable and robust communication paradigm for sensor networks, *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networks (MobiCOM '00)*, Boston, USA, August 2000, 56-67.
- [17] **Heinzelman, W.R. Chandrakasan, A. and Balakrishnan, H.**, 2000. Energy-Efficient Communication Protocol for Wireless Microsensor Networks, *Proceedings of the 33rd Hawaii International Conference on System Sciences*, Maui, Hawaii, January 2000, Volume 8, 10-19.
- [18] **Hellerstein, J., and Hong, W.**, 2003. TinyDB: In-Network Query Processing in TinyOS, Sam Madden, *TinyOS Document*, **Version 0.4 September 2003**, Intel Research and Berkeley, California, USA.

- [19] **Madden, S. Franklin,M.J. and Hellerstein,J.M. Hong,W.,**2002. TAG: a Tiny Aggregation Service for Ad-Hoc Sensor Networks, *5th Annual Symposium on Operating Systems Design and Implementation (OSDI)*., USA December 2002.
- [20] **Hong,W.,**2003. Tiny Application Sensor Kit (TASK) *TinyOS Document, TinyOS 1.1* ,Intel Research, Berkeley, USA
- [21] **Yao,Y. Gehrke, J. E,** 2003. Query Processing in Sensor Networks. *First Biennial Conference on Innovative Data Systems Research (CIDR 2003)*, Asilomar, California, January 2003.
- [22] **Internet Draft,** 1999. draft-ietf-manetaadv-04.txt., Ad hoc on demand distance vector (AODV) routing October 1999.. *Perkins, C.*
- [23] **Sadagopan,N. Krishnamachari,B. and Helmy,A.,**2003. The ACQUIRE Mechanism for Efficient Querying in Sensor Networks, *First IEEE International Workshop on Sensor Network Protocols and Applications (SNPA) in conjunction with IEEE ICC 2003*, Anchorage, AK, USA ,May 2003,**PAGES.**
- [24] SeMA project home page, 2003. <http://ics.yeditepe.edu.tr/tnl/html/sema.html>.
- [25] **Baydere, S. and Ergin, M.A.,** 2002. An architecture for service access in mobile ad hoc networks, *Proceedings of IASTED WOC*, Banff, Canada, July 2002, 392-397.
- [26] **W3C Recommendation,** 2000. <http://www.w3c.org> Extensible Markup Language (XML), Bray,T. Paoli, J. and McQueen, C.M.S.
- [27] **Baydere, S. Ergin, M.A. and Durmaz, Ö.,** 2003 A unifying architectural view for sensor networks in environmental monitoring, draft submitted to Kluwer Mobile Networks and Applications.
- [28] **W3C Draft,** 2003. XQuery 1.0: An XML Query language (*work in progress*), Boag, S. Chamberlin, D. Fernandez, M. F. Florescu, D. Robie, J. and Simeon,J.
- [29] **Karp, R. Elson, J Estrin,D. and Shenker,S.,**2003. Optimal and Global Time Synchronization in Sensornets, *CENS Technical Report, 0012*, UCLA, California, USA.

- [30] **Elson, J. and R'omer, K.**, 2003. Wireless Sensor Networks: A New Regime for Time Synchronization. *ACM SIGCOMM Computer Communication Review (CCR)*, 33(1), 149–154.
- [31] **Hill, J. Szewczyk, R. Woo, A. Hollar, S. Culler, D. E. and Pister, K. S. J.**, 2000. System architecture directions for networked sensors, *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, November 2000, 93-104.
- [32] Crossbow Technology Inc., 2004. <http://www.crossbow.com>
- [33] **Gay, D. Levis, P. Behren, R.von Welsh, M. Brewer, E. and Culler, D.**, 2003. The NesC Language: A Holistic Approach to Networked Embedded Systems, *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, San Diego, California, USA, June 2003,1-11.
- [34] **Levis, P. Lee, N. Welsh, M. and Culler, D.**, 2003. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Application, *Proceedings of the first international conference on Embedded networked sensor systems* Los Angeles, California, USA 2003, 126-137.
- [35] **Edward,S. and Hollar,A.**,1996. COTS Dust, 1996, *MSc Thesis*, University of California, Berkeley, USA.
- [36] **Crossbow Technology, Inc.**, 2003. MPR/MIB Mote User Manual, *User's Manual, Document 7430-0021-05*, San Jose, CA, USA.
- [37] **von Eicken, T., et al.**,1992. Active messages: a mechanism for integrated communication and computation, *19th Annual International Symposium on Computer Architecture*, Queensland, Australia, May 1992.
- [38] **Kernighan, B. W. and Ritchie, D. M.**, 1988. The C Programming Language, Second Edition. Prentice Hall.
- [39] **Koc, M.**, 2003. MoTeS: Monitoring Tool for Networked Sensors, *BSc Thesis*, Yeditepe University, Istanbul, Turkey.

- [40] **Ye, W. Heidermann, J. and Estrin, D.**, 2003. Medium Access Control with Coordinated, Adaptive Sleeping For Wireless Sensor Networks *USC/ISI Technical Report, ISI-TR-567*, USC/Information Sciences Institute,USA.
- [41] **Crossbow Technology, Inc.**, 2003. MTS/MDA Sensor and Data Acquisition Boards User's Manual, Rev B, San Jose, CA
- [42] **Mainwaring, A. et al**, 2002. Wireless Sensor Networks For Habitat Monitoring, *Proceedings of ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, Georgia, USA,2002,88-97.
- [43] **Elson,J. Bien, S. Busek, N. Bychkovskiy, V. Cerpa, A. Ganesan, D. Girod, L. Greenstein, B. Schoellhammer, T. Stathopoulos, T. and Estrin D.**, 2003. EmStar: An Environment for Developing Wireless Embedded Systems Software ,*Technical Report ,CENS Technical Report 009*, UCLA,California,USA
- [44] **Ganesan, D. Govindan, R. Shenker, S. Estrin, D**, 2001. Highly Resilient,Energy Efficient Multipath Routing in Wireless Sensor Networks, *Proc. ACM MOBIHOC '01*,USA,October 2001, 251-253
- [45] **Levis, P. Lee, N**, 2003. TOSSIM: A Simulator for TinyOS Networks, *Technical Document, September 17*, UCB, California, USA.
- [46] **Gay, D. Levis, P. Culler, D. Brewer, E.**, 2003. NesC 1.1 Language Reference Manual, May **2003**, UCB, California, USA.
- [47] **Hill, J. L.**2003. System Architecture for Wireless Sensor Networks, *PhD Thesis*, Computer Science in University Of California, Berkeley, USA.
- [48] **Levis,P. Patel,N. Shenker,S. and Culler,D.**,2003. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks, *UC Berkeley Tech Report, UCB//CSD-03-1290*, UCB,CA,USA.
- [49] **Crossbow Technology, Inc.**, 2003.Mote In Network Programming User Reference, *TinyOS document* <http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf> , San Jose, CA, USA.

APPENDIX A. EMSTAR

Increasing research attention has been directed towards wireless sensor networks. As the community moved into more complex design efforts a number of important software design issues have arisen. Therefore, the need for good quality simulators has become important. Simulation of wireless sensor networks is a difficult task to achieve since modeling wireless communication channel and sensing interfaces is very difficult. Another problem is with the software design: in order to save energy cross layer protocols are becoming more obvious today and the interactions between the layers can cause unpredicted problems in the real life. EmStar [43] is a Linux based software framework developed by University of California, Los Angeles. EmStar's novel execution environment encompasses pure simulation, true in-situ deployment, and hybrid mode that combines simulation with real wireless communication and sensors situated in the environment. Each of these modes run the same code and use the same configuration files, allowing developers to seamlessly iterate between the convenience of simulation and the reality afforded by physically situated devices.

EmStar is an attempt to balance the usefulness of a simulator with the need to write software that works in reality. EmStar allows developers get the basics of an algorithm working in a controlled environment (simulation); then understand both the effects of the real environment via the ceiling array and portable arrays.

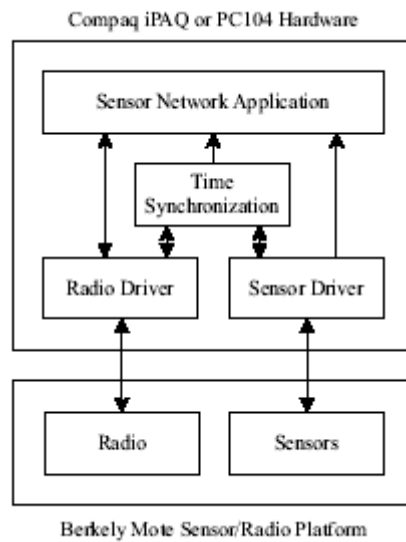


Figure A.1 EmStar Software Stack

In a real deployment, autonomous and untethered nodes are deployed in a real environment, running a real application. Each node has a low power radio and sensors, and runs an EmStar software stack. EmStar software stack includes device drivers that provide interfaces to real physical channels such as network and sensing interface. In Figure A.1, the architecture of a simple base station with EmStar software stack is given. PC runs EmStar server side tools and user application and sensor device runs EmStar sensor software stack. Drivers provide the connection between the application and sensor device.

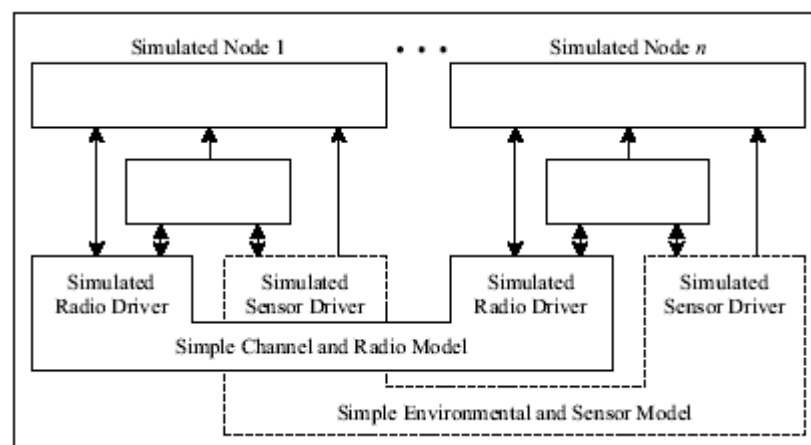


Figure A.2 EmStar Simulator Architecture

In the simulation mode, the architecture of the software stack changes at the driver level as seen in Figure A.2. Instead of communicating with real hardware, simulated radio and sensor drivers act as if the application has received response from real hardware. Each instance of the code runs a different process in the memory and by

using RPC mechanisms between processes drivers are emulated. The main advantage of the simulator is that nodes running simulation can easily log distributed events in their global temporal order and pc side tools such as debuggers, memory check tools can be used to debug simulations.

The ceiling array feature allows simulations to be done using real hardware. All the instances of the node stack are run centrally. However, the channel simulator module is not used; instead, each simulated node is mapped to one of the motes connected to the server. When a node sends a packet, it is transmitted and received by real motes, through the real channel. As a result, the environment causes real distortion and multipath fading effects to the simulation environment.

Having the ability to communicate with real sensing interfaces and radio devices, the sensor inputs can be recorded and later these recorded time series are played back in real time to a simulation environment. EmStar provides a hybrid environment for simulation and is an essential tool to be used for complex environments. Interactions between simulated nodes and real nodes are possible and each simulated node can run different application. Ceiling array provides the adverse effects of real life to the simulation. Consequently, most of the problems can be solved by using this environment.

APPENDIX B. IN NETWORK PROGRAMMING

Xnp [49] is a Crossbow implemented network programming toolkit included in TinyOS. Network programming is to load and transfer the program code using wireless communication rather than direct communication to PC host. Several approaches to in-network programming exist. Although XNP is a limited version of in-network programming, it was the first to achieve this. Trickle [48] is an enhanced version of XNP that provides scalability and it focuses on multi hop reliable program code distribution.

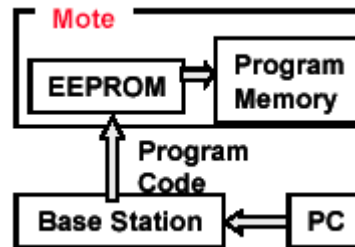
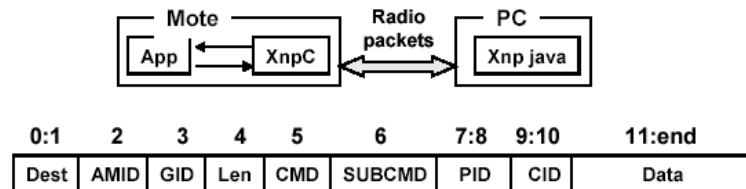


Figure B.1 Network Programming

Network programming works mainly in two stages. First the program is stored outside the program memory through radio packets. Second the downloaded code is transferred to the program memory and the mote reboots with the new code.

Network programming consists of mote modules and a java program on PC host. Between a mote and PC, messages of reserved message ID (47) are transferred. The format of the message is shown in Figure B.2.



- AMID: set to #47.
- CMD: type of command (e.g., download, query and etc).
- PID: Checksum for program code. Used for validation
- CID: Sequence number for capsule

Figure B.2 Xnp Network Programming Protocol

Network programming starts with download start message. After sending start of download message a couple of times, PC side program sends each line of program as a capsule. Sensor node receives these capsules and stores them in EEPROM. Once transmission of capsules finishes, PC side application sends download terminate message to notify the end of download. Then, the mote searches any missing capsule in its EEPROM and asks for retransmission of it to PC side. This operation continues until there aren't any missing capsules left. After download of missing capsules are completed, mote transfers the image to its program memory and restarts with new application.

AUTOBIOGRAPHY

I was born in Tokat at 27 March 1978. I graduated from Istanbul Bahcelievler Adnan Menderes Anatolian High School at 1996 and had my BSc degree from Control and Computer Engineering Department of Istanbul Technical University at June 2000. I graduated from the University at 2000 and enrolled for degree of MSc of Computer Engineering at Istanbul Technical University same year. I have worked in TUBITAK-UEKAE for 3 years as a researcher while having my MSc Education. I've worked as a research student at The Network Laboratory of Yeditepe University while preparing my MSc Thesis. By the completion of my MSc thesis, I've started working in StMicroelectronics in April 2004. My areas of interest include distributed systems, real time operating systems, embedded systems, internet routing protocols and wireless systems.