DYNAMICALLY CONTROLLABLE APPLICATIONS FOR
WIRELESS SENSOR NETWORKS

By

Sriram Rajan

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Engineering
in the Department of Electrical and Computer Engineering

Mississippi State, Mississippi

May 2006
DYNAMICALLY CONTROLLABLE APPLICATIONS FOR
WIRELESS SENSOR NETWORKS

By

Sriram Rajan

Approved:

Georgios Lazarou
Assistant Professor of Electrical and
Computer Engineering
(Major Advisor)

Nicholas H. Younan
Professor of Electrical and Computer
Engineering
Graduate Coordinator

Thomas Philip
Professor of Computer Science and
Engineering
(Committee Member)

Yul Chu
Assistant Professor of Electrical and
Computer Engineering
(Committee Member)

Roger L. King
Associate Dean for Research and Graduate
Studies
Applications for Wireless Sensor Networks can be updated dynamically by means of wireless upgrade mechanisms. Current research efforts in wireless upgrade mechanisms for WSN have focused on transmitting application packets for upgrades via wireless medium. However, these schemes require significant overhead involved in sending and receiving application packets that affect the sensor operation, in addition to bringing the nodes down to reprogram and restart them.

By designing applications in a way that allows dynamic functionality changes during operation, the overhead and sensor delays can be eliminated. Dynamically Controllable Application (DCA) is a novel scheme for designing WSN applications whose behavior can be rapidly and dynamically changed during operation. The results indicate that a veritable functionality change is achieved in a span of a few milliseconds.
DEDICATION

I would like to dedicate this research to my parents, Mr. D.N.T. Rajan and Mrs. Geetha Rajan and my sister, Ms. Suchitra Rajan.
ACKNOWLEDGMENTS

This thesis would not have been possible without the guidance of my major advisor, Dr. Lazarou, who has inculcated in me several values to work aggressively on research problems. His patience in my work encouraged me to work persistently in my research until the results were finally attained. I also thank my boss, Mr. Hannigan, for his support and encouragement throughout the course of my study.

I wish to extend my thanks to many people in the TinyOS community for helping to resolve several of my programming problems.

I could not find any words to thank my friends Arun Ramakrishnan, Ashwini Mani, Gaurav Marwah and several others for both motivating me in my research and assisting me in reviewing several versions of my thesis.

Also, I thank my committee members for patiently reviewing my thesis and suggesting improvements at various stages. Finally, I must acknowledge the motivation and support received from my friends, which has helped my work at every stage.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEDICATION</td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENTS</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>LIST OF ABBREVIATIONS and NOMENCLATURE</td>
<td>xi</td>
</tr>
<tr>
<td>CHAPTER</td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Statement and Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Summary of Main Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Organization</td>
<td>4</td>
</tr>
<tr>
<td>II. BACKGROUND AND RELATED WORK</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Wireless Sensor Networks</td>
<td>5</td>
</tr>
<tr>
<td>2.2 TinyOS Components and Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.3 TinyOS Scheduler Execution and Interrupt Priority</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Application Architecture</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Concurrency Model of TinyOS</td>
<td>13</td>
</tr>
<tr>
<td>2.6 Using TOSSIM Simulator for Evaluation</td>
<td>14</td>
</tr>
<tr>
<td>2.7 Related Work in Application Layer Enhancements</td>
<td>15</td>
</tr>
<tr>
<td>2.7.1 Using Simplicity to Control Complexity</td>
<td>15</td>
</tr>
<tr>
<td>2.7.2 Reliable Upgrades for Group Communication Software</td>
<td>16</td>
</tr>
<tr>
<td>2.7.3 A Framework for Live Upgrade of Software</td>
<td>17</td>
</tr>
<tr>
<td>2.7.4 Survivable Network Applications</td>
<td>18</td>
</tr>
<tr>
<td>2.8 Comparison of Related Research Efforts</td>
<td>19</td>
</tr>
<tr>
<td>2.9 Previous Research Attempts</td>
<td>19</td>
</tr>
<tr>
<td>2.9.1 Directed Diffusion with Live Upgrades</td>
<td>19</td>
</tr>
<tr>
<td>2.9.2 Continuous Data Transmissions During Upgrades in WSN</td>
<td>20</td>
</tr>
<tr>
<td>III. DYNAMICALLY CONTROLLABLE APPLICATIONS</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>3.1 Designing Applications to be Dynamically Controllable</td>
<td>25</td>
</tr>
<tr>
<td>3.1.1 Message Structure</td>
<td>25</td>
</tr>
<tr>
<td>3.1.2 DCA Application Abstraction and Control</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3 Message Transmission and Reception</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Application Pseudocodes</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 Design of Applications with Controlled Periodic transmissions and State Reset Operations</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1.1 Example of a Static Application with Periodic Transmission</td>
<td>30</td>
</tr>
<tr>
<td>3.2.1.2 Example of a DCA Application with Controlled Transmission Frequency</td>
<td>31</td>
</tr>
<tr>
<td>3.2.1.3 Example of a Static Application with State Initiation at Startup</td>
<td>32</td>
</tr>
<tr>
<td>3.2.1.4 Example of DCA application with State Control Mechanism</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2 Design of Applications with Different COTS Modules</td>
<td>33</td>
</tr>
<tr>
<td>3.2.2.1 Example of Applications using COTS Components</td>
<td>35</td>
</tr>
<tr>
<td>3.2.2.2 Example of DCA Application that uses Interchangeable COTS Components</td>
<td>37</td>
</tr>
<tr>
<td>IV. VALIDATION AND EXPERIMENTS</td>
<td>39</td>
</tr>
<tr>
<td>4.1 Validation and Experimentation Outline</td>
<td>39</td>
</tr>
<tr>
<td>4.2 Validation and Testing Metrics</td>
<td>41</td>
</tr>
<tr>
<td>4.3 Experimental Setup</td>
<td>42</td>
</tr>
<tr>
<td>4.3.1 Hardware Setup for Implementation of Application</td>
<td>42</td>
</tr>
<tr>
<td>4.3.2 Software Simulation of Application and Motes</td>
<td>45</td>
</tr>
<tr>
<td>4.4 Experiments with DCA scheme</td>
<td>46</td>
</tr>
<tr>
<td>4.4.1 Case I: Experiments with Controllable Reset</td>
<td>46</td>
</tr>
<tr>
<td>4.4.2 Case II: Experiments to Verify a Frequency Change within an Application</td>
<td>49</td>
</tr>
<tr>
<td>4.4.3 Case III: Experiments Involving Changing Executing Module Based on User’s Commands</td>
<td>51</td>
</tr>
<tr>
<td>4.5 Discussion</td>
<td>55</td>
</tr>
<tr>
<td>V. CONCLUSION</td>
<td>56</td>
</tr>
<tr>
<td>5.1 Conclusions</td>
<td>56</td>
</tr>
<tr>
<td>5.2 Problems Encountered in this Research</td>
<td>57</td>
</tr>
<tr>
<td>5.3 Future Work</td>
<td>58</td>
</tr>
</tbody>
</table>
CHAPTER

REFERENCES .......................................................................................................................... 60

APPENDIX

A. TINYOS CODE USING DCA SCHEME ........................................................................ 63
LIST OF TABLES

TABLE                                      Page
2.1  Example of a TinyOS application operation sequence................................. 11
3.1  Structure of a sample control message packet .............................................. 25
3.2  Periodic message transmission application.................................................. 30
3.3  Example of DCA application with controllable frequency.............................. 31
3.4  Psuedocode comparison of a static application with state transitions
    and initial state initiation............................................................................ 32
3.5  Example of DCA application with state control............................................. 33
3.6  Comparison of static applications using different COTS components
    A and B respectively.................................................................................... 36
3.7  Example of DCA psuedocode for transition between different COTS
    components .................................................................................................... 38
4.1  Results obtained from implementing software reset operation
    for achieving synchronization........................................................................... 48
4.2  Structure of variable change control message packet.................................... 50
4.3  Comparison of DCA mechanism with Incremental Upgrades [20] scenario .. 51
4.4  Structure of a module change control message packet ................................ 52
A.1  Command interface file “Command.nc”............................................................ 64
A.2  Command definition file “CommandM.nc” ....................................................... 65
A.3  Message types and other structure information ............................................... 66
<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.4  Simulation scripts for GUI: left column (tython), right column (TOSSIM autorun)</td>
<td>67</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Functions of elements involved in a sensor node</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>An illustration of a WSN topology</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Structure of a sample messaging component</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Sequence diagram showing split-phase operation in Sensor Network</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Application Framework in TinyOS [18, 21]</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Simplified arbiter mechanism based on the Simplex architecture [4]</td>
<td>15</td>
</tr>
<tr>
<td>2.7</td>
<td>Sequence of Live Upgrade Framework [9]</td>
<td>18</td>
</tr>
<tr>
<td>2.8</td>
<td>Example of continuous data transmission, Data messages (type =4) are transmitted with upgrade messages (type= 163)</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>The block diagram of the DCA scheme, downward arrows indicate commands, upward arrows are events signaled</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>Proposed flexible design for TinyOS Application</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>DCA scheme operation for frequency/state change operation</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>Flow chart of DCA application for interchangeable COTS modules</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Phases in TinyOS Application Design, Testing and Deployment</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Experimental setup for hardware evaluation</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>MessageCenter application</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Software simulation setup</td>
<td>45</td>
</tr>
<tr>
<td>FIGURE</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>4.5</td>
<td>Energy consumption in millijoules (mJ) for grid layout N = 4, m = 70 seconds.</td>
<td>48</td>
</tr>
<tr>
<td>4.6</td>
<td>Energy consumption in milliJoules (mJ) N=100, n=2, lossy network with radio source.</td>
<td>53</td>
</tr>
<tr>
<td>4.7</td>
<td>Energy consumed in millijoules N=100, n =2, lossy network with LED output source and radio inactive</td>
<td>54</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Full Form</td>
<td></td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Network</td>
<td></td>
</tr>
<tr>
<td>TinyOS</td>
<td>TinyOS platform for sensor networks</td>
<td></td>
</tr>
<tr>
<td>TOSSIM</td>
<td>Near real-time simulator for TinyOS applications</td>
<td></td>
</tr>
<tr>
<td>Flash</td>
<td>Program flash, used for storing executable program in the sensor node</td>
<td></td>
</tr>
<tr>
<td>DCA</td>
<td>Dynamically Controllable Application</td>
<td></td>
</tr>
<tr>
<td>LED</td>
<td>Light Emitting Diodes</td>
<td></td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read Only Memory</td>
<td></td>
</tr>
<tr>
<td>RFM</td>
<td>Radio module</td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>Processing in the order of “First In”, “First Out” or based on the arrival</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
<td></td>
</tr>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
<td></td>
</tr>
<tr>
<td>PSFQ</td>
<td>Pump Slowly Fetch Quickly</td>
<td></td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
<td></td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
<td></td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
<td></td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versioning System</td>
<td></td>
</tr>
<tr>
<td>COTS</td>
<td>Common Off The Shelf</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER I

INTRODUCTION

An approach for an early Tsunami warning system is now possible with the use of sensor nodes [1]. This approach could save thousands of lives across the world if ever another Tsunami were to occur. In the wake of recent Hurricanes, there has been an increased interest in the use of Wireless Sensor Networks (WSN). These networks can be deployed across any terrain and are instantly operational [2].

Modern day advances in wireless sensor technology have made remote monitoring of hostile locations such as dense forests, ocean depths, and desert areas more feasible, due to the reduced costs of the nodes and the aid of more effective communication protocols [5, 6, 17].

Each of these nodes consists of a particular application that achieves a specific function. For example, an algorithm that implies “transmit reading if light intensity is greater than 4 units” could be compiled into an application. This application is then loaded into the motes\(^1\) via hardware connection or by means of wireless upgrades [20, 21].

Current methods of application deployment in the area of wireless programming [8, 20, 21] allow these sensors to be programmed with an application even during their

\(^1\) The term “Sensor node” is commonly referred to as “mote” or “node” and these terms can be used interchangeably.
field operation, irrespective of its location. The advancement in this field of rapid application deployment has an impact on the sensor operation. The upgrade (wireless programming) applications require transmitting several packets of newer application information. Further, the upgrade protocol requires overhead on each of the sensor nodes involved in either sending or receiving the upgrade data, in addition to performing its sensor function.

In this thesis, a novel scheme is presented that enables applications to change their sensing functionality during run-time without requiring system downtimes as in the case of conventional upgrades. This scheme utilizes the TinyOS concurrency approach to design dynamically controllable applications. We validated and evaluated the performance of our Dynamically Controllable Application (DAC) design scheme by performing experiments on a small scale wireless sensor network. We also developed and executed simulation experiments using TOSSIM [24]. The results demonstrate sensing functionality changes within a few milliseconds.

1.1 Problem Statement and Motivation

The main requirements for implementing dynamically controllable applications are [8, 11]:

1. Propagation: The packets\(^2\) must reach all of the nodes in the network for efficient implementation.

\(^2\) Packets are short messages transmitted by the nodes in the sensor network and received by all other nodes. Each packet contains source and destination addresses. A node processes a packet only if it is the destination node or if the packet was broadcast to all the nodes [18]
2. *Lifetime of the network:* The overall lifetime of the network must not be affected by the implementation. The data traffic in the sensor network must be minimal for effective communication within the sensor network.

3. *Limited power availability:* Any implementation must consider the limited power availability and must restrict the number of control data transmissions. The mica2 motes [2] used in this research ran on a couple of ‘AA’ type battery cells that can not support long term use of some sensor applications [10].

In Deluge [21], the propagation criterion is met by providing multihop transmissions. However, the required excessive upgrade mechanism traffic reduces the overall network lifetime. The work in [20] proposes a scheme based on incremental upgrades that meets the last two design requirements. However, it does not meet the first requirement, *propagation.*

Results from preliminary experiments with Deluge [21] indicated that application executions can be performed concurrently during upgrade operations. Based on this concurrent behavior, the reliable application framework of Deluge, as suggested in [4], can be applied to the TinyOS architecture. That allowed us to develop the architecture of our Dynamically Controllable Application (DCA) design scheme that enables the design of statically designed components, whose interconnections and operations are defined prior to compilation of the application, as dynamically controllable.
1.2 Summary of Main Contributions

In this thesis, a DCA design scheme is developed that exploits the concurrency mechanism of the TinyOS platform to dynamically control the behavior of wireless sensor network applications. Applications designed using DCA attain the following objectives:

- Interchangeable modules at run-time;
- Controllable frequency of sensor operations, such as transmitting readings or performing actions at regular intervals;
- Ability to control the mote operations by terminating/resetting a mote to a particular state during operations.

1.3 Organization

The thesis has been organized as follows.

Chapter II discusses the background of sensor networks and its operation along with providing examples of various concepts in TinyOS platform. The nesC language is also described in detail.

Chapter III presents the Dynamically Controllable Application scheme in detail.

Chapter IV discusses the details of the Experimental setup and test cases considered. The results obtained are also presented here.

Chapter V concludes the research with possible avenues for future work.
CHAPTER II
BACKGROUND AND RELATED WORK

2.1 Wireless Sensor Networks

Wireless Sensor networks (WSN) are composed of a large number of sensor nodes or motes. These motes are connected via a wireless medium to a base station that transmits sensor readings such as light, temperature, and humidity to users. An outline of the components and their respective functions in a sensor network is shown in Figure 2.1.

![Diagram of Wireless Sensor Network Components]

Figure 2.1 Functions of elements involved in a sensor node
A sensor node consists of the following components:

- Processor (for processing sensor data)
- Sensor device
- Transreceiver (for wireless transmission and reception)
- Peripheral devices such as LED’s, Flash, EEPROM

A sensor application is therefore a combination of software and hardware that accomplishes a specific task [7]. This software-hardware interface is discussed in detail in Section 2.4. An illustration of a WSN for implementing a sensor application is shown in Figure 2.2.

Figure 2.2  An illustration of a WSN topology
**Communication Interface:** A “Communication Interface” serves as a link between the host computer and the base station.

**Base Node:** (connected to base station) A base node provides the base station the processing capability to receive, process and transmit messages from the sensor nodes to the host computer.

**Base station:** The base station, unlike other sensor nodes, is connected to the external power supply and is also used to program the base node if it needs re-programming. When combined with a base node the base station acts as a central node that communicates with the host computer (through the communication interface) and can transmit control or data signals to all nodes in the network.

**Sensor Node:** These are sensor devices, capable of wireless communication and sensor data processing. As discussed in [17], wireless sensor networks are capable of withstanding random node losses due to obstacles or relocation in the hostile environment where they are deployed. Sensor nodes maintain communication with the base and might be able to communicate with one or more other sensor nodes. Inter-node communication is critical in a scenario where multi-hop communication (base cannot reach all nodes directly) is desirable. In a multi-hop scenario a sensor node would assist the base node by forwarding messages from other unreachable nodes to the base station.

### 2.2 TinyOS Components and Architecture

TinyOS is the most widely used platform for programming sensor networks because of its modular nature and its extensive support of platforms. Therefore, TinyOS has been chosen for application development in this thesis.
The TinyOS platform has been designed using the nesC language [25], which is based on C language. The various elements of TinyOS are:

- **Event**: Events can be called by signals or commands, and are capable of signaling other events. These events preempt tasks.

- **Task**: Tasks are generally posted by events and commands. They cannot preempt other tasks and usually run to completion, unless preempted by other events.

- **Commands**: Commands are non-blocking, return status, and can call lower level events.

- **Calls**: Calls are used for initiating events.

- **Posts**: Posts can be used for calling tasks, and are preempted by event calls.

- **Interface**: Interface provides a component and implements its functions. For example: `Main.StdControl -> SenseToInt`; causes an interface of one component to point to that of another component.

- **Module**: A module consists of one or more interfaces and either uses or defines the events and tasks in that interface.

- **Component**: A component comprises of events, modules, operation states, and interfaces. A component is a state machine for the sensor application as it contains a thread for execution. It also provides event handlers and is capable of signaling events. The various elements of a component are shown with a sample component in Figure 2.3.

- **Configuration**: Each application consists of a main configuration file that “wires” or connects all other components together with interfaces from other components.
Figure 2.3  Structure of a sample messaging component.

- **Split-phase operations**: To facilitate simpler design of low latency TinyOS applications, some of the operations are performed in split-phase. Figure 2.4, shows an illustration of this operation, where the processor requests sensor data and exits.

  In Figure 2.4, a module implements the operation and signals when the data is ready. This split-phase operation allows the processor to simply call the function and execute other events while waiting on the status of the transmission.
The nesC language [25] has many elements that separate the definition (such as events and tasks) from the interfaces (interconnection between components). It provides library of “Common Off The Shelf” (COTS) components that can be used by the TinyOS programmer. The main function performed by the TinyOS architecture is to initialize the hardware and enable the interrupts. Once these are accomplished the scheduler is called to schedule events and tasks to perform operations on the TinyOS hardware.

The scheduling takes place in an endless loop and can only be terminated by lower level system interrupts [10].
2.3 TinyOS Scheduler Execution and Interrupt Priority

The execution of a TinyOS Application follows the pattern as shown in Table 2.1

Table 2.1 Example of a TinyOS application operation sequence.

<table>
<thead>
<tr>
<th>Main ()</th>
<th>(\Rightarrow) start</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hardware_Initialize ()</td>
</tr>
<tr>
<td>While (1)</td>
<td>{</td>
</tr>
<tr>
<td>scheduler();</td>
<td>(\Rightarrow) Organizes task and event execution, can be bypassed by other events</td>
</tr>
<tr>
<td>run_next_task();</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

A key feature of this architecture is the facility for non-blocking execution. The execution begins with the initialization of the TinyOS hardware. Hardware initialization is followed by the execution of the scheduler that schedules the execution of the tasks and events. The execution of hardware events always preempts that of any other event handler or task. Tasks are not allowed to preempt one another or other events. Therefore, a task or event always runs to completion. However, the occurrence of an event moves the related event handler up the list of scheduled events preceding any other task or event handlers present in the list. The TinyOS application scheduler model uses a simple and efficient scheduler that prevents application deadlocks and other race conditions [10, 11]. Also, the scheduler allows for efficient concurrent execution of events and tasks as detailed in section 2.5.
2.4 Application Architecture

A TinyOS application is a graph connecting the various components [7]. The block diagram of a component is shown previously in Figure 2.3. Figure 2.5 represents the various components that comprise a complete TinyOS application.

Application: This layer is responsible for the sensor’s behavior and contains all of the component interconnection graphs that link various components to appropriate modules. This layer can also be used to specify fixed routes using simple algorithms based on the node id such as “If node id is 5, Parent is node 1”.

GenericComm: This component provides the necessary interfaces to send and receive messages and also signals when the messaging is complete.

AMStandard: This component extends the function of the GenericComm component by adding two main functionalities: Ability to send packets via radio (using RadioCRCPacket) and ability to send data via computer’s serial port (UARTPacket). The data is then broken down into bytes for serial port transmission. For transmission using radio, these bytes are further decomposed to bits and are transmitted using the hardware interface.

Byte level components: The components UART Byte (provides data to the computer), RadioByte, Photo, and Temperature interact directly with the hardware elements and send/translate data.

Bit level components: These components involve the RFM, the Clocks, temperature, and light sensor devices.
2.5  **Concurrency Model of TinyOS**

The concurrency model in TinyOS is enforced by means of run-to-completion tasks and interrupts. The scheduler component usually runs these tasks in FIFO order in accordance with the run-to-completion rule, unless interrupted by events that precede the tasks during execution.

![Application Framework in TinyOS](image)

**Figure 2.5** Application Framework in TinyOS [18, 21].
The interrupt handler has a much higher priority over the scheduler and can preempt the operation of scheduled tasks and events. Due to this reason, nesC compiler statically links components via the interfaces. This helps to avoid race conditions by the use of atomic statements. Further, the nesC compiler identifies and reports these data races to the programmer during compile time. Two types of codes are possible:

**Synchronous Code:** These consist of functions, Commands, Events, and Tasks.

**Asynchronous Code:** Using *atomic* keyword in events and tasks and creating *async* events allows effective resource sharing between components. It is implied that although an event can interrupt any task or event, the sections of code within tasks marked atomic, or those events which are designated as *async*, cannot be interrupted and form the only exceptions to the rule [25].

### 2.6 Using TOSSIM Simulator for Evaluation

TOSSIM [24] is a tool that enables the execution of large-scale simulation experiments of sensor motes in real-time. Additionally, with the capacity of power profiling recently added to this architecture, it provides a platform for evaluation of the power consumption of the sensor application, prior to its deployment.

Several research works have used TOSSIM successfully in comparison to *ns-2* for sensor network evaluation [21, 26]. TOSSIM’s main advantage over *ns-2* is its ability to implement the operations at network level. Further, applications designed in TOSSIM have high probability of running successfully on actual hardware motes, as stated in [24].

Several other advantages of TOSSIM are its ability to provide high fidelity simulation for TinyOS applications, allowing for network monitoring and packet
injection. Further, applications simulated in TOSSIM match their counterparts implemented in actual sensor node. Several plugins are available for use with TOSSIM for monitoring debug messages.

2.7 Related Work in Application Layer Enhancements

2.7.1 Using Simplicity to Control Complexity

Lui Sha [4] analyzed the complexity between Fault Avoidance and Fault tolerant modes of implementation and showed that it is simpler to correct faults than to design a fault avoidance mechanism. The main components of this architecture are shown in Figure 2.6.

![Simplified arbiter mechanism based on the Simplex architecture](image)

Figure 2.6 Simplified arbiter mechanism based on the Simplex architecture [4].

Reliability of a system can be improved by using a simple and reliable core component that ensures the system’s critical functions, despite failure of other non-
critical components. The key to the success of this architecture is that the development of the high-tolerance controller must satisfy the requirement: impact caused by incorrect actions must be tolerable and recoverable [4]. This simplex architecture has been designed to trade usability for more reliability.

2.7.2 Reliable Upgrades for Group Communication Software

The research by P.V. Krishnan et al. [3] has resulted in a reliable mechanism for communication software upgrades in sensor networks. This architecture consists of two modules: the experimental module and the safety module. The safety module is capable of handling the most critical functions that are required for the basic operation of the system. The experimental module is used to implement the new experimental communication protocol for specified network application. This architecture has been based on the model illustrated in Figure 2.6.

In addition, the common faults are monitored at regular intervals to determine the stability of the system. Some examples described in [3] are:

1) Software errors resulting in malformed messages such as:
   a. Messages with invalid id;
   b. Messages never sent by a valid sender.

2) Improper data delivery due to error in communicator or system protocol stack are caused by random message losses and delays experienced by one or more members.
The threshold level determines the stability of the system and is determined by the conformance to critical functions required of the system. The architecture allows for transition from experimental to safety module during operation based on this threshold.

To perform the testing, several scenarios are analyzed with faults introduced on purpose to test the systems response in detecting these faults. For each of the tests the percentages of accuracy, false alarm and right decision are determined. The tests have given an indication of high performance that can be attained using the above architecture.

### 2.7.3 A Framework for Live Upgrade of Software

The research by Lizhou, *et al.* [9] proposes a framework for a live software upgrade mechanism using a modular programming approach. The mechanism suggested provides a means for upgrading applications during run-time without interrupting their operation.

A Module proxy component is used that maintains the state of each module and also allows for concurrent upgrades. The modules are dynamic and can be replaced.

An instance of the new module is created and its handle is registered with the version control manager. The module enters into a quiescent state for upgrade and the incoming requests are buffered. The present state is saved in a shared memory and the Module proxy associates with the new module implementation. The various sequences of operations are depicted in Figure 2.7.
The old module is unloaded from the memory, and a suitable mechanism for concurrent execution is chosen that reduces service downtime of the application. A single component is chosen as master and the others are slaves. The operation is achieved as a series of state transitions, which are started by the Upgrader (Master) and conclude at the slaves.

2.7.4 Survivable Network Applications

The first requirement of a survivable network application is the ability to anticipate and avoid a failure scenario and implement a solution in the framework at the code level. This research case study by Robert, et al [15] suggests a mechanism for
designing survivable applications. In this approach, there is a four-step process for developing the resultant survivable network application. These steps outline the basic needs for ensuring reliable operation that includes specification, analysis for possible intrusions, determining critical functions, and providing these critical functions during intrusions/failures.

These steps are followed by ample case studies as examples for developing specifications based on usage scenarios, developing intrusion scenarios, and providing means for resisting these intrusion scenarios by strengthening the current mechanisms.

2.8 Comparison of Related Research Efforts

The work in [3] by Prasanna et. al requires a Linux environment to implement the operations for the SPIN architecture used in this scheme. Therefore, this scheme cannot be adapted for the limited TinyOS architecture.

Work in [4, 9, 15] apply to general software design principles and have been used in this research.

Motivated by similar work in [19, 21] for wireless upgrade schemes, the need for light overhead scheme with minimum data traffic was felt necessary.

2.9 Previous Research Attempts

2.9.1 Directed Diffusion with Live Upgrades

Directed Diffusion [12] provides means for efficient communication between the node requesting data (source) and a node providing data (sink). Some of the main
features provided by Directed Diffusion protocol include path reinforcement, node healing and sustenance of the network.

This work sought to implement the Directed Diffusion [12] application in order to achieve live upgrades in sensor networks and minimize the communication time involved. An algorithm was obtained and the coding started for sensor networks platform. However, further enhancements to the Directed Diffusion mechanism were not conceived except for a project type implementation. Therefore, instead of implementing a project, this work was abandoned.

2.9.2 Continuous Data Transmissions During Upgrades in WSN

The authors in [21] prioritized data dissemination and upgrade mechanism to continuous sensor operation. The concurrent application execution of both sensor application and data transmission protocols was therefore difficult. This limited the data that the upgrade source could transmit during the upgrade operation. The operation of the receiver node was also affected significantly during the upgrade process.

To overcome this limitation, the following application architecture was conceived. This application is a modification of the work performed in [4] to transmit sensor readings at periodic intervals. The goal of the experiment was to test whether or not this event interrupt would be performed within the upgrade operation.

Figure 2.8 shows a listing of output messages received at the base station starting with the timestamp of the form “hours : minutes : seconds . milliseconds”. The timestamp field is followed by the addr or address field to which the message is addressed (65535 signifies broadcast) and the group field specifies the group to which the
motes belong. The *group* field is necessary for separating outputs from multiple groups of sensor networks. The following *type* field is the key to identifying the nature of the message. Each component typically uses its own message type (sensor readings, node status messages) for data transmission. These are followed by the *length* and the *data* fields, which denote the “data-length” and the “actual sensor data” respectively.

![MessageCenter](image)

**Figure 2.8** Example of continuous data transmission, Data messages (*type* =4) are transmitted with upgrade messages (*type* = 163).

This experiment has shown that this limitation can be overcome by embedding events within the components that allow for controlled events and restricted data
transmission. From the implementation of the continuous data transmission scheme, data
transmission was attained prior to each upgrade packet. However, the process of
embedding the lines of code within the component and interconnecting the necessary
interfaces is proved to be too complex to generalize. Hence, this approach of executing
multiple operations was extended to form the basis of the Dynamically Controllable
Applications scheme presented in the following chapter.
CHAPTER III
DYNAMICALLY CONTROLLABLE APPLICATIONS

The main functionality of the Dynamically Controllable Application (DCA) component is to trigger an event after receiving a message from the base station. This would, in turn, issue a command to the sensor application to dynamically re-configure itself. Figure 3.1 shows the block diagram of an application using DCA.

Figure 3.1 The block diagram of the DCA scheme, downward arrows indicate commands, upward arrows are events signaled.
The DCA abstraction permits the modules to interact with the messaging and radio components directly. The control mechanism issues *start/stop* signals for either of the modules in the application. The DCA mechanism is transparent to the user as it interacts with both the user and the application. Therefore, the DCA abstraction allows user to control the module that interacts with the output interface.

*Frequency*\(^3\) *control:* This component of the DCA scheme contains all the necessary functions and variables that control the frequency of various operations in a TinyOS application. This is required because sensor applications typically use periodic logging or other sensor data reporting functions. Controlling this periodicity is a necessary requirement to restricting the data traffic of the application.

*State Control:* The state control component of the DCA scheme is identical to the frequency control. In addition, the state control involves additional variable settings and declarations based on the current and next state.

*Module Control:* The need to change the working module\(^4\) by allowing the architecture to support two working modules was motivated by [3, 4, 21]. This is achieved by diverting the sensor output to other interfaces that are statically available in the application.

\(^3\) Frequency refers to the rate of activity, such as transmission, in this context. This variable controls the frequency of Timers that are used to trigger either data transmission, or some other processing on the sensor data.

\(^4\) Module and COTS component refer to the same entity and have been used interchangeably.
**Modules:** The DCA scheme can support and control two or more modules in the application. These modules are typically COTS components that can be used with stand-alone code as well (without DCA scheme).

**Communication and Messaging Interface:** This interface receives the messages from the base node and forwards appropriate event calls to the internal modules and to the DCA sub-layer. The DCA uses unique message types provided by TinyOS to distinguish its command messages from other types of messages received by the application modules.

**Hardware and Radio Components:** The various components involved in these blocks are responsible for Radio and Serial communication, Clocks and other Sensor components.

### 3.1 Designing Applications to be Dynamically Controllable

#### 3.1.1 Message Structure

Table 3.1 provides the required message structure for the DCA mechanism.

<table>
<thead>
<tr>
<th>Field</th>
<th>Source (16 bytes)</th>
<th>Action (8 bytes)</th>
<th>Value (8 bytes)</th>
<th>Module (16 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Address of the source node</td>
<td>Simple action code for each unique action</td>
<td>Frequency of operation and other values</td>
<td>This field decides which module is to be activated.</td>
</tr>
</tbody>
</table>

The operation of accepting command messages is based on the readily available *SimpleCmd* application. The functionalities were scaled down for the requirements of
this DCA mechanism. These reduced functionalities support the conversion of the received message to appropriate nesC function calls.

### 3.1.2 DCA Application Abstraction and Control

As discussed in Chapter 2, the events have a higher priority than tasks and preempt their operations. The concurrency mechanism of TinyOS allows the user to implement multiple functionalities by calling mutually exclusive events.

The design mechanism of TinyOS requires for static linking of all “Common Off The Shelf” (COTS) components or modules prior to the compilation of the application. A mechanism that allows for linking multiple COTS components by abstracting the output interface was felt necessary. Therefore, this output interface should be independent of the component that uses its commands and events. A sequence of operations for this desirable flexible TinyOS design is shown in Figure 3.2.

![Figure 3.2 Proposed flexible design for TinyOS Application](image URL)
It is possible to design an application by the use of SimpleCMD, whose components and the application variables are alterable during field operation. The sequence of operations required for designing controllable applications are:

*Application Outline:* This phase requires that the application be designed for a specific function, such as: “to broadcast sensor data at periodic intervals”.

*Design of DCA:* This design mechanism takes into consideration the need for the application to be dynamically controllable. It creates functions that modify the variables/application states and might provide interchangeable modules as required. These functions are defined to work as event calls, which are triggered when command messages are received, and seek to interrupt the running task in the application if possible.

*Use of SimpleCMD for controlling application behavior:* The SimpleCMD application needs to be modified and included with the original application. The necessary modification would allow the SimpleCMD to post event calls directly to the event handlers in the DCA application that perform the variable/module changes.

### 3.1.3 Message Transmission and Reception

The DCA scheme transmits messages by broadcast/multicast/node address, as specified, and uses the modified Simple CMD application receives control messages to control DCA application behavior. With the message structure as defined, a small Java program can be written that transmits data over the communication medium discussed earlier. Example codes are provided in Appendix A.
3.2 Application Psuedocodes

The following pseudocode comparisons between conventional design methodology and DCA highlight the optimizations attainable by using DCA. These codes are discussed with their respective flow charts.

For simplification from the original TinyOS code, the following functions have been used in the pseudocode:

- **Read variable**: Reads a value from the included header files and stores it in a variable.
- **Timer**: This function is based on the Timer component available with TinyOS [10]. The functions Timer.start (variable) and Timer.stop () are used to start/stop or the timer interface.

3.2.1 Design of Applications with Controlled Periodic transmissions and State Reset Operations

DCA allows for implementing controlled periodic transmissions and State changes within applications during their field operation in an identical manner as that of the original application. While the functions are identical, the process of state change requires some additional variables and processing operations to be performed. To further illustrate this operation, example applications without DCA are illustrated with appropriate pseudocodes. The required sequence of operations to achieve a controlled periodic operation or state change via DCA mechanism is shown in Figure 3.3.
The operation initializes its components similar to any TinyOS application. Then it performs the essential functions of component initiations and sensor data transmissions at the required frequency. The decision operations of the loop are mainly provided by the command reception and processing section of the SimpleCMD variant used in this program. The frequency and state control are linked directly to the appropriate DCA
event handlers provided in the DCA application. The process of *Resume Operation* is shown to illustrate the flow of the code as controlled by the scheduler. The following examples are provided for further understanding.

### 3.2.1.1 Example of a Static Application with Periodic Transmission

In this example, a static application that transmits data periodically at a preset frequency is considered. A psuedocode for this application is shown in the Table 3.2.

#### Table 3.2 Periodic message transmission application

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Read Transmit_Frequency</code></td>
<td>Variable Transmit_Frequency is set to transmission frequency in milliseconds.</td>
</tr>
<tr>
<td><code>Timer.start (Transmit_Frequency, REPEAT)</code></td>
<td>Repeated Timer</td>
</tr>
<tr>
<td><code>While (Not(Stop_Request()))</code></td>
<td>Continuous loop operation until terminated</td>
</tr>
<tr>
<td><code>If Timer.Fired()</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>Transmit (value)</code></td>
</tr>
<tr>
<td><code>End While</code></td>
<td></td>
</tr>
</tbody>
</table>

A basic sensor application that reads a variable and configures its Timer interface accordingly is shown in the psuedocode. This application repeats the Timer operation until the termination or stop commands are executed, from within or by the scheduler. However, there is no provision for modifying the frequency variable and restarting the Timer interface.
3.2.1.2 Example of a DCA Application with Controlled Transmission Frequency

DCA functionality has been added to the periodic transmission pseudocode as shown in the Table 3.3. The DCA application is capable of controlling the transmission frequency without interrupting the periodic transmission of the sensor node.

Table 3.3 Example of DCA application with controllable frequency

<table>
<thead>
<tr>
<th>Read Transmit_Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer.start (Transmit_Frequency, REPEAT)</td>
</tr>
<tr>
<td>While (Not(Stop_Request()))</td>
</tr>
<tr>
<td>If Timer.Fired()</td>
</tr>
<tr>
<td>Transmit (value)</td>
</tr>
<tr>
<td>If Frequency.change(New_Frequency) ........ (\rightarrow) Variable change request received</td>
</tr>
<tr>
<td>Timer.stop()</td>
</tr>
<tr>
<td>Timer.start(New_Frequency,REPEAT) .. (\rightarrow) Frequency reset operation completed by restarting Timer with new frequency.</td>
</tr>
<tr>
<td>End While</td>
</tr>
</tbody>
</table>

This functionality change is achieved by connecting the readily available SimpleCmd application to the “Frequency.change (variable)” function in this application. The component interconnection in TinyOS allows a this component to call a function in another application. The DCA implementation specifies the “Timer.stop( )” function to stop the currently running Timer component and “Timer.start(New_Frequency, REPEAT)” to restart the Timer and specify its repetitive nature.
3.2.1.3 Example of a Static Application with State Initiation at Startup

A simple TinyOS application that implements a simple state transition is considered. This static application uses Timer interface to control its state transitions. The triggering of a sensor event by the Timer component controls the transition of states from the initial state, ‘1’ to another state, ‘2’. At this instant, the Timer interface is terminated, as it is required only once during the operation of the application. A psuedocode for this implementation of state transition based on Timer component is shown in Table 3.4.

Table 3.4 Psuedocode comparison of a static application with state transitions and initial state initiation

| Initialize State variables() .......... | ➔ Declares and defines state variables |
| Read Transmit_Frequency            |                                      |
| While (Not(Stop_Request()))       |                                        |
|   If Timer.Fired()                 |                                        |
|     State_Transition (value) .......... | ➔ Waits for Timer event to initiate state transition |
| End While                          |                                        |

Although, a simple state transition is achieved from this application, the desired frequency change during operation is not possible. Further, as the application executes the Timer component only once, a transition between states is not possible after the first timer initiation. This operation would require some event to be triggered to interrupt the flow of the application.
3.2.1.4 Example of DCA application with State Control Mechanism

In the original implementation (without DCA), it was required to restart the entire application to implement the state change functionality. With the DCA application, a state change is easily implemented by passing a suitable command message to the application. The psuedocode for an application implementing this required DCA functionality is shown in Table 3.5.

Table 3.5 Example of DCA application with state control

<table>
<thead>
<tr>
<th>Initialize State variables ()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Transmit_Frequency</td>
</tr>
<tr>
<td>While (Not(Stop_Request()))</td>
</tr>
<tr>
<td>If Timer.Fired()</td>
</tr>
<tr>
<td>State_Transition (value)</td>
</tr>
<tr>
<td>If State.change(new_state)</td>
</tr>
<tr>
<td>Re_initialize_state_variables() ....... Simple function to reset variables required for state transition</td>
</tr>
<tr>
<td>End While</td>
</tr>
</tbody>
</table>

From the DCA psuedocode, it is inferred that the operations for state initiation (at application startup) and those for state changes in DCA are separate functions. Therefore, the DCA implementation has reduced processing overhead during state change operations. In comparison to restarting and defining all of the variables in the original application, the DCA scheme alters only a few required variables.

3.2.2 Design of Applications with Different COTS Modules

The sequence of operations in this mechanism is shown by means of a flowchart in Figure 3.4.
Figure 3.4  Flow chart of DCA application for interchangeable COTS modules
In the DCA implementation, various modules are initialized and then the DCA mechanism takes over from there. The user can send command messages using the MessageCenter interface or write a simple Java program to send console messages to the Base node. These commands are used to control the various COTS modules allowed by the DCA scheme. Based on the commands, the DCA scheme uses function calls to implement module changes. Since these are triggered by event calls, they preempt the currently running task on each sensor node. Therefore, the DCA can arbitrate over the application functionality and allows a certain level of control over the application. The functions are capable of stopping and starting the respective modules when required. Once started, the applications and the modules run non-stop until all of the available power is consumed or termination signal is received.

3.2.2.1 Example of Applications using COTS Components

Table 3.6 shows a psuedocode for two applications that use different COTS components A and B respectively. These components are independent of each other but use the same output interface to obtain output values to be transmitted (via radio) or displayed (via leds).
Table 3.6  Comparison of static applications using different COTS components A and B respectively

<table>
<thead>
<tr>
<th>COTS Component A</th>
<th>COTS Component B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Transmit_Frequency</td>
<td>Read Transmit_Frequency</td>
</tr>
<tr>
<td>COTS_Module_A.initialize()</td>
<td>COTS_Module_B.initialize()</td>
</tr>
<tr>
<td>COTS_Module_A.start()</td>
<td>COTS_Module_B.start()</td>
</tr>
<tr>
<td>Output = COTS_Module_A</td>
<td>Output = COTS_Module_B</td>
</tr>
<tr>
<td>While (Not(Stop_Request()))</td>
<td>While (Not(Stop_Request()))</td>
</tr>
<tr>
<td>Display_Output()</td>
<td>Display_Output()</td>
</tr>
<tr>
<td>End While</td>
<td>End While</td>
</tr>
<tr>
<td>COTS_Module_A.stop()</td>
<td>COTS_Module_B.stop()</td>
</tr>
</tbody>
</table>

The psuedocodes gives example of two applications using different COTS components. The following functions are required for each application:

- **Initialization**: Initialization of the components is required prior to its usage or implementation.
- **Start/Stop**: These commands prepare the component for regular use and to access the functions provided by the appropriate interface.
- **Output**: Output is essentially a mapping between output functions. TinyOS allows linking between these components. The declaration allows local access to the remote function, as though the function were explicitly declared within the application.
- **Display_Output**: The function Display_output sends output to the chosen output interface.
3.2.2.2 Example of DCA Application that uses Interchangeable COTS Components

Previously, applications would need to be redesigned, recompiled, and loaded to the sensor nodes in order to implement this change in functionality. The DCA implementation reduces the processing overhead involved by eliminating the need for redesigning the application. An example of a DCA application that uses interchangeable components is shown in Table 3.7.

Applications that use the DCA scheme can easily change from one COTS component (COTS_Module_A) to another (COTS_Module_B) provided these were included in the design phase. The output interface functions independently of the component used and transmits the output data to the required component as defined.

In this chapter the DCA scheme has been presented. The DCA scheme uses the concurrency feature of TinyOS to its advantage in designing applications. These applications can be changed rapidly while the sensors are in field operation. Features such as frequency, state, and working modules can be changed dynamically if the application is designed using DCA scheme. The various cases of applications and related psuedocodes in DCA have been analyzed. These psuedocodes would need to be implemented in TinyOS for further evaluation as performed in the following chapter.
Table 3.7  Example of DCA pseudocode for transition between different COTS components

```
Read Transmit_Frequency

COTS_ModuleA.initialize() .................\(\rightarrow\) Component initiation required for all components
COTS_ModuleB.initialize()

COTS_ModuleA.start()
Output = COTS_ModuleA

While  (Not(Stop_Request()))
    If Timer.Fired()
        Transmit (value)
    If Module.change()
        COTS_Module_A.stop() .................\(\rightarrow\) Start/ Stop Commands to control components
            COTS_Module_B.start()
        Output = COTS_Module_B ...............\(\rightarrow\) Output changed from one interface to another
End While
```
CHAPTER IV
VALIDATION AND EXPERIMENTS

4.1 Validation and Experimentation Outline

In this section, the DCA mechanism has been validated and results are obtained from the experiments. Then the results are compared with identical scenarios in WSN upgrade protocols [20, 21]. The design and testing scheme for typical TinyOS applications has been outlined in Figure 4.1.

![Figure 4.1 Phases in TinyOS Application Design, Testing and Deployment.](image-url)
The nesC compiler performs the first testing and validation operation for any TinyOS application. It checks for potential data races and warns the programmer if any were observed [25]. This is followed by a hardware evaluation on a few motes with alternating LED’s\(^5\) for diagnostic displays. Finally, a thorough evaluation with human-readable debug messages is made possible by using the TOSSIM simulator. The results are stored in a data file that can be parsed for any errors or failure in code operation. These failures must be anticipated during the design of the application, and debug messages must be provided for evaluation. A main advantage of using TOSSIM is that several areas of the code can be tested, which might not be possible with a simple LED display.

After the results are obtained, from hardware setup, the results from an appropriate simulation of a lossy sensor network model [24] are taken for comparison. It might be necessary to repeat hardware or software simulations for various changes in environmental conditions or random node failures.

An analysis of the performance of the proposed application design scheme is done to determine the number of errors or message losses during transmission. The TOSSIM simulator allows a better analysis of errors and message losses from code-level rather than its functionality verification in hardware setup. These results are compared to a predetermined loss threshold for acceptable operation. If acceptable, the application is

---

\(^5\) Each mote consists of three LEDs namely red, green and yellow corresponding the bit values. ‘1’ represents ‘on’, ‘0’ represents ‘off’. For instance red LEDs would be ‘on’ and others ‘off’ for value (4)\(_{10}\) or (100)\(_2\).
deployed to field motes. Otherwise, the application debug messages obtained are used to isolate the erroneous code and redesign the application.

4.2 Validation and Testing Metrics

The works in [21, 26] have also made use of the TOSSIM simulator and their evaluation schemes can be analyzed for a better understanding of the simulations. Based on the work in [20, 21, 22, 26] the following metrics were found necessary for validating the DCA scheme:

- **Time to Completion:** This metric corresponds to the speed of the entire operation to completely affect the entire network. This factor plays a key role in determining how the nodes respond to losses in transmission due to interferences in wireless medium, as discussed in [18]. Mate [22] presents a similar analysis of the time taken for complete network injection of its program capsules.

- **Verification of operation:** This stage is critical for any TinyOS application. In general programming practice, debug messages are critical in locating code faults and correcting them. However, due to the limited size of messages transmitted from TinyOS hardware, it is preferred that these debug messages are generated in a simulator during program design. Further, more humane messages such as “Message transfer failed!” are useful in isolating errors to that specific condition that caused this message. However, a program might follow different execution paths based on the hardware platform or simulator that it specified during compilation. Therefore, alternating between the
hardware implementation and simulation is inevitable to confirm the operation of the application.

- **Comparison of results:** Related approaches in implementing application changes such as the work in [20, 21] have provided comparable results to this implementation. These works serve as a motivation to evaluate and compare the DCA approach.

- **Testing:** This validation operation is where the results obtained are further analyzed for fault conformance. A low fault margin is a general indication of the application’s suitability for field implementation. However, the location of the faults needs to be determined and the faulty code must either be removed or corrected.

### 4.3 Experimental Setup

The experiments were conducted in two phases to separate the hardware and the simulation results. The hardware setup involves programming the sensor nodes to test the functionality of the desired application and latency in operation. Other factors affecting the operation of sensor networks such as message losses, code failures are observed from the software simulation. The setup and implementation of these two phases are discussed below:

#### 4.3.1 Hardware Setup for Implementation of Application

This phase consists of a single base mote running *TOSBase* application. This application forwards the messages with its own group id back to the computer via the
MIB510CA serial gateway [10]. Other nodes in the network were involved in transmitting or receiving the sensor information. The entire simulation was performed with one or more source and multiple destination nodes as shown in Figure 4.2.

The following equipments were utilized for this research:

1. Linux/Cygwin loaded with TinyOS 1.1.10
2. MIB510 Programming board [2]
3. Mica 2 motes (433 Mhz) [2]
4. Multimeter (to measure battery voltages and to replace the batteries if necessary)
5. Soldering Iron for fixing mote connections mainly with battery.

Figure 4.2   Experimental setup for hardware evaluation.

Another requirement for all the experiments is the use of an appropriate application that can transmit the required control messages from the host computer to
base station in the specified format. The *MessageCenter* application achieves this functionality as illustrated in Figure 4.3.

![MessageCenter application](image)

**Figure 4.3**  *MessageCenter* application

The *MessageCenter* application readily adapts to any new message structures defined or created in TinyOS when started. Then, it allows a user to send control messages to both the hardware setup (via base node) and to simulations (using TOSSIM interface) easily.
4.3.2 Software Simulation of Application and Motes

In the simulation setup, illustrated in Figure 4.4, with node 0 turned off. This node is used as a virtual base node that sends control messages to all other nodes in the network. Over 100 nodes have been successfully simulated for running any TinyOS application. Grid layout is chosen as this allows the developer to specify random internode losses by using the LossyBuilder application. These values are generated for varying network size with fixed distance between the nodes from 2 to 20 feet as performed in [21].

![Software simulation setup](image)

Figure 4.4 Software simulation setup

The number of nodes (apart from Base node) denotes the size of the sensor network, and is denoted by \( N \). The node spacing for the simulation is specified in feet and has been denoted by \( n \). Finally, the total simulation time is denoted by the letter \( m \).
It should be noted that the MessageCenter is not the only means of controlling simulations as simple scripts can be used as shown in Table A.4, in the Appendix.

The results obtained from simulation were analyzed for the following factors:

- Verification of results: This is necessary to correlate the results provided by the simulation to those obtained from the sensor hardware.
- Energy consumption: Component-wise energy estimation needs to be performed using the plug-ins with TOSSIM [24] simulator.
- Losses in transmission: Losses in transmission need to be determined as they are critical to estimating the performance of any TinyOS application.

4.4 Experiments with DCA scheme

In this section, the DCA mechanisms for application modifications are designed based on the architecture suggested in Chapter 3. These individual mechanisms are tested using available hardware, and further simulations are performed in the TOSSIM simulator [24] as necessary.

4.4.1 Case I: Experiments with Controllable Reset

The main purpose of this Case of experiments is to compare implementation of software reset in a Traffic light application using DCA application with an identical operation involving node restart via upgrade. The reason for performing a reset operation is to evaluate effectiveness of the DCA mechanism in a new traffic application scenario. Eleven experiments were performed to evaluate state changes in sensor applications using the DCA scheme. In the first 10 sets of experiments, the starting sequence was
altered on hardware motes and the final experiment was performed using simulation. The messages were injected using the MessageCenter application, and the results of synchronization after the reset operation were observed for various cases. This experiment uses only the source and action fields of the control message packet discussed earlier with their default values of 0 and 1 respectively. In this experiment, a simple Traffic light mechanism is implemented with the following sequence on motes with respective id’s from 1 to 4:

1. Initially the mote’s respective yellow LED’s are blinking. This state is termed as state 1.

2. Once a synchronization message is received from every other node, the traffic light operation begins with a delay based on the mote id, to transition to state 2. For instance, if the delay for state 2 is set at 4 seconds, node 1 will begin in 4 seconds, node 2 in 8 seconds and so on.

3. At any instant, all of the motes can be forced into the synchronizing state or state 1 by means of a reset signal from the base.

The results obtained from the final set of experimental simulations are illustrated in 0. The difference between these hardware and software results is seen from Table 4.1.
Table 4.1 Results obtained from implementing software reset operation for achieving synchronization

<table>
<thead>
<tr>
<th>Mode</th>
<th>Synchronization time in seconds</th>
<th>Re-synchronization time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>35</td>
<td>16</td>
</tr>
<tr>
<td>TOSSIM</td>
<td>23</td>
<td>45</td>
</tr>
</tbody>
</table>

In the first 10 cases, a few cases were observed where the synchronization could not take place due to collisions and other errors. However, once the reset signal was applied, the synchronization took place instantly. Due to the unique node-synchronization requirement, No identical implementation with upgrade protocols was found for comparison. From our experiments, we found that the performance improvement would be obvious as is shown in the following experiments. Further, these experiments validated the working of the Traffic Light application.

Figure 4.5 Energy consumption in millijoules (mJ) for grid layout N = 4, m = 70 seconds.
From the results obtained, the software simulation in experiment 11 did not resynchronize as effectively as the hardware, though the results were concurrent with the expected flow of the code discussed in Chapter 3. The delays experienced in software could be attributed to random node losses within the network. Since the Traffic light application has a unique node-synchronization requirement, a message loss caused distinct delays in this application. From the experimental observations, it was clear that these errors were constant, based on the size of the simulated network. This was confirmed with a linear increase in power requirements, based on time taken for experimentation.

4.4.2 Case II: Experiments to Verify a Frequency Change within an Application

The main purpose of this experiment was to compare changing frequency of operation using DCA scheme with that of manual update of operation frequency and reloading application via upgrade protocols [20, 21]. Two sets of experiments were performed to verify frequency change in the application. The first experiment was on hardware motes, which was followed by verification via simulation. These experiments are implemented by allowing a variable change in the program from the command line or via the Message Center application. The command structure required for the experiments is shown in Table 4.2. The frequency field has been added for the communication payload. The value of frequency needs to be specified in seconds. The software simulation on TOSSIM was performed on a combination of 100 nodes in a matrix of 10x10 nodes (grid layout).
The experiments were successful and the software simulation has shown that the transition in frequency timers was extremely precise. There results from the hardware and software simulation were found to be identical. Therefore, the hypothesis that the DCA implementation is functional has been verified.

For the simulation experiment, a readily available SenseToRfm application has been modified using DCA to adjust the frequency of sensor transmissions. The command signal was transmitted via base node. The addition in code complexity and overall time to completion were the main metrics used for evaluation. The savings in time and performance over that of an upgrade protocol such as [20, 21] is shown in Table 4.3.
Table 4.3  Comparison of DCA mechanism with Incremental Upgrades [20] scenario

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Time to complete</th>
<th>Code complexity Added in lines</th>
<th>Packet transmission mechanism</th>
<th>Power source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 2</td>
<td>Insignificant (few ms on Hardware)</td>
<td>98</td>
<td>From base node/ on demand</td>
<td>External (from base)</td>
</tr>
<tr>
<td>Incremental Upgrades</td>
<td>25 seconds</td>
<td>&gt; 100</td>
<td>Individual nodes /periodic</td>
<td>Internal (within node)</td>
</tr>
<tr>
<td>Deluge</td>
<td>23 seconds</td>
<td>2</td>
<td>Individual nodes /periodic</td>
<td>Internal</td>
</tr>
</tbody>
</table>

The power requirement for data transmission was found to be linear, increasing steadily with the simulation time. At any instant, all of the nodes consumed the same amount of energy down to their individual components. The losses were observed to be consistent and only varied with the network size.

4.4.3 Case III: Experiments Involving Changing Executing Module Based on User’s Commands

The hypothesis that two or more modules of the application could be swapped during the operation using DCA scheme needed factual evidence. The main aim of this case of experiments was to compare this DCA scheme (if hypothesis is verified) with that of manually updating the application configuration and initiating the upgrade via upgrade protocol in [21]. Experiments were performed on both hardware and via simulations to prove the proposed hypothesis. The main aim of these experiments was to evaluate a functionality change by shifting the output from one node to another. This evaluation was followed by comparison with the upgrade protocol [21].

---

6 Based on the results observed in [20]
7 Experiments performed for monitoring minimum data transmission of one packet
These experiments use two available applications namely SenseToLeds and SenseToRfm. These applications are capable of independently transmitting sensor data either over Leds or Radio respectively. A transition from one component (Radio) to another (Led) during application operation would be critical to assessing the effectiveness of the proposed hypothesis. The hardware setup involved studies on 8 physical motes with one base node. A software simulation was also performed on 100 nodes. The message structure as shown in Table 4.4 has been used for transmitting messages in these experiments.

Table 4.4  Structure of a module change control message packet

<table>
<thead>
<tr>
<th>Source (16 bytes)</th>
<th>Action (8 bytes)</th>
<th>Module (16 bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>‘1’ for Radio or ‘2’ for LED</td>
</tr>
</tbody>
</table>

The results from hardware setup showed as the nodes made a distinct transition from transmitting sensor data on Radio to displaying the same on LED’s. This simple observation proves the hypothesis from the hardware experiments.

An output of the final simulation experiment is shown in Figure 4.6. As mentioned earlier the simulation experiments have been repeated across a variety of node inter-spacing (from 2 feet to 20 feet apart) with varying loss-levels (generated using LossyBuilder application). The results have shown consistency across each of these experiments in terms of energy consumption and losses observed. Also, the transition of output from Radio to Led was implemented instantly, confirming the hypothesis.
The energy consumption of a sensor network with LED as the dominant node is shown in the Figure 4.7. This shift in energy consumption confirms the change in the executing module. The difference in energy levels for the radio is not significant as is noted in these statistics. This could be attributed to the idle listening state of the radio module. From the experiments, it was confirmed that this did not affect the output from the LED’s and therefore adjusting the receiver power level has been left for future work.
The cases used for this experimentation have also been implemented via the upgrade protocol developed in [21]. The main factors of comparison are:

- **Time to complete operation:** varies with the size of the network. For a network of 10 nodes, the hardware setup showed an average of 1 minute. In contrast, the entire setup of the dynamic module replacement experiment lasted for 20 seconds even during simulation purposes.

- **Downtime:** is a huge factor as most of the nodes are either in their reprogramming or restarting phases. Typically, it lasts for up to 10 seconds during the upgrade process from the experimental setup. However, the DCA application was successful in implementing the module change without affecting the sensor operation.
4.5 Discussion

In this chapter, an experimental outline for DCA has been discussed and suitable examples from related work have been chosen for both implementations of the experiments as well as analysis of the results. In each of the experiments performed a hypothesis has been presented, and has been defended with the relevant results. From the results obtained it is clear that:

1. A DCA scheme can be effectively implemented and tested using any TinyOS application.
2. The energy consumed does not vary significantly for interchanging the modules during application operation.\(^8\)
3. The operation of the sensor application remains unaffected by the DCA implementation.\(^9\)
4. The time to completion for DCA is lesser than that for achieving identical operation in upgrade protocols [20, 21].\(^{10}\)

\(^8\) From Figure 4.7
\(^9\) Results from comparison in Section 4.4.3
\(^{10}\) Experimental results shown in Table 4.3
CHAPTER V
CONCLUSION

5.1 Conclusions

In this thesis, a new scheme (DCA) is presented that uses command-parsing interfaces to allow a user to bypass the static application development in sensor networks.

By means of using the DCA scheme, selected features can be modified more rapidly than with the current upgrade protocols [19, 21]. For instance in Case 1, it is possible to reset the state of an application for node synchronization as in a Traffic Light implementation, Case 2 allows the adjustment of the node transmission frequency. Finally, Case 3 effectively implements a modular change for controlling the output. With these cases, the overall load in a system has been significantly reduced without affecting the node operation downtime. Therefore, the use of Dynamic Application Design scheme is recommended for providing efficient applications that can be dynamically controlled from the console.

From the results obtained, the DCA mechanism was found to be more time saving, in the order of few milliseconds, compared to several seconds to minutes required for implementing similar changes via upgrade protocols.
5.2 Problems Encountered in this Research

Some of the major issues involved in this thesis effort, that could minimize future efforts are listed below, in the order of their priority:

1. **Reliable applications for sensor networks**: Most of the applications used for this thesis were based on CVS versions or freshly developed code, as they were made available. This resulted in an occasional inconsistency in the results. For instance, in estimating the energy consumption, all of the results could not be provided in a textual format for further processing. The workaround used for this problem was overcome to pause the simulation at the end and to scroll down the graphical results obtained. Alternatively, the interface could be redesigned to write some of the power values to a file with a simple java code.

2. **Lack of plotting functions**: More effective plotting functions could be designed that can trace the growth of application variables with time.

3. **Determining failed transmissions in sensor nodes**: With the data obtained from the simulation the number of failed nodes was constant for a given network size. In each of these cases, the range was varied from 2 to 20 feet with random inter-node losses (specified in a loss file). However, from the manufacturer’s datasheet [2], the range of the mica2 motes was obtained as 500 ft. Simulations could be extended around this range to determine if additional results can be obtained.
5.3 Future Work

In this thesis, an exhaustive evaluation with the existing upgrade schemes such as *Incremental Upgrades* [20] and *Deluge* [21] was not performed, as working simulations of these applications could not be obtained. An evaluation could assist in the development of a suitable architecture for providing upgrades that can be designed based on the DCA scheme. Dynamically changing applications using reset signals, transmission frequency changes and module changes achieve reduction in the number of upgrade events in the network during run-time. Also, the frequency of upgrades can be effectively controlled for reducing the data traffic in the entire network.

Our research shows that DCA scheme is highly effective in implementing a systematic change in the application behavior without the use of an upgrade protocol. Some of the main requirements of this upgrade protocol using DCA are:

1. The protocol should be simple enough, allowing easy adaptation with existing sensor applications.
2. A component wise analysis should be performed to reduce the cases of upgrades. This can be achieved either by implementing switching between interfaces prior to the loading the application or by changing application variables such as thresholds.
3. The operation must ensure that the minimum node functions are effectively provided.
4. The node idle time must be effectively used for the completion of upgrade.
The overall time taken and the sensor node activity, during the period of the upgrade, would be the most important parameters for evaluating of this upgrade protocol. Also, a complete evaluation of faults during the transmission would be required for varying distances and random loss values between the nodes.
REFERENCES


APPENDIX A

TINYOS CODE USING DCA SCHEME
The design of the configuration file for the command interface is shown distinctly in Table A.1. The structure of the command message is specified in the file AM_CMDMSG and has been discussed later.

Table A.1 Command interface file “Command.nc”

```plaintext
includes IntMsg;

configuration Command{
    provides interface ProcessCmd;
}
implementation {
    components Main, CommandM, IntToLedsAndRfm, GenericComm as Comm;

    Main.StdControl -> CommandM;
    ProcessCmd = CommandM.ProcessCmd;
    CommandM.Comm -> Comm;
    CommandM.ReceiveMsg -> Comm.ReceiveMsg[AM_CMDMSG];
    CommandM.FreqControl -> IntToLedsAndRfm.FreqControl;
    CommandM.SourceControl -> IntToLedsAndRfm.SourceControl;
}
```

The definition file used for providing function, event and task definitions for the above Command file is shown in Table A.2. The command results and their status messages and the resultant signals are described in detail.
Table A.2 Command definition file “CommandM.nc”

```plaintext
// $Id: CommandM.nc
includes IntMsg;

module CommandM
{
    provides {
        interface StdControl;
        interface ProcessCmd;
    }
    uses {
        interface StdControl as Comm;
        interface ReceiveMsg;
        interface FreqControl;
        interface SourceControl;
    }
}

implementation {
    TOS_MsgPtr cur_msg;
    TOS_Msg buf;

task void CommandInterpret () {
    struct CmdMsg *cmd = (struct CmdMsg *) cur_msg->data;
    
    result_t status = SUCCESS;

    // do local packet modifications:
    cmd->source = TOS_LOCAL_ADDRESS;

    // Execute the command
    switch(cmd->action) {
    case RESET:
        // Check if any value was sent if so,
        // call FreqControl
        
        break;
    case CHANGE:
        call SourceControl.change_module((uint8_t)cmd->setmodule);
        break;
    case NONE:
        break;
    default:
        status = FAIL;
        
        signal ProcessCmd.done(cur_msg, status);
    }

    command result_t StdControl.init() {
        cur_msg = &buf;
        return (call Comm.init());
    }

    command result_t StdControl.start() {
        return SUCCESS;
    }

    command result_t StdControl.stop() {
        return SUCCESS;
    }

    command result_t ProcessCmd.execute(TOS_MsgPtr pMsg) {
        if (returnval == SUCCESS)
            // with the required value
            if (cmd->value == 0)
                call FreqControl.reset(0);
            else
                call FreqControl.reset((uint16_t)cmd->value);
            break;
    case CHANGE:
        call SourceControl.change_module((uint8_t)cmd->setmodule);
        break;
    case NONE:
        break;
    default:
        status = FAIL;
        
        signal ProcessCmd.done(cur_msg, status);
    }

    command result_t StdControl.init() {
        cur_msg = &buf;
        return (call Comm.init());
    }

    command result_t StdControl.start() {
        return SUCCESS;
    }

    command result_t StdControl.stop() {
        return SUCCESS;
    }

    command result_t ProcessCmd.execute(TOS_MsgPtr pMsg) {
        if (returnval == SUCCESS)
```
The various variable declarations and message structures are shown in the Table A.3 along with the constants for state declarations.

### Table A.3 Message types and other structure information

```c
enum {
    AM_LiveDataMsg = 164,
    AM_CMDMSG = 165,
};

enum {
    LIVE_RELIEABLE_DELAY = 500,
    LIVE_STATE_DELAY = 4000,
    LIVE_INTERSTATE_DELAY = 2500,
    NUMBER_MOTES = 4,
    TRANSMIT_MIN_DELAY = 10
    TRANSMIT_MAX_DELAY = 100,
};

typedef struct LiveDataMsg {
    // Live Data message structure
}
```
Table A.3 (continued)

```c
STATE_MAX_DELAY=2,
};
enum {
    uint8_t yellow;
} LiveDataMsg;
```

// Command message structure
typedef struct CmdMsg{
    uint8_t action;
    uint16_t source;
    uint16_t value;
    uint8_t setmodule;
} CmdMsg;

Finally, the simulation script used in this research is shown in Table A.4

Table A.4  Simulation scripts for GUI: left column (tython), right column (TOSSIM autorun)

```python
# This script is in the public domain and has no copyright
#
#
# Sriram : needs CmdMsg.class file in net.tinyos.tools folder

from simcore import *
import os
import re
#from simtime import *

from net.tinyos.tools import CmdMsg
msg=CmdMsg()
msg.set_action(2)
```

```
numsec 65
nummotes 5
precmd echo "Running a grid of 4 motes"
layout grid
export DBG=usr2
executable build/pc/main.exe -p
logfile log-grid-5.txt
postcmd "End of 1st simulation"
```

```
numsec 65
nummotes 5
precmd echo "Running a random set of 4 motes"
layout random
export DBG=usr2
executable build/pc/main.exe -p
logfile log-random.txt
```
Table A.4 (continued)

```python
msg.set_setmodule(1)
start = 2

def repeat_sim(spacing, nodes):
    args='-p -b=0 -l=1.0 - '+
    rf="../Experiments/10x10-"+
    str(spacing)+'.nss" -t=20'
    lossfile ="./LossExptsWithPower/log-
    10x10-"+str(spacing)+'.txt"
    print "\n Args = " +args +"n" + "
Lossfile = "+lossfile
    nodeid = 1

    try:
        sim.dumpDBG (lossfile)
    except:
        print "\n Simulation failed because of existing logfile"
        sim.exit()

    sim.exec ('build/pc/main.exe', nodes+1
                ,args)

    motes[0].turnOff()

    while nodeid < nodes+1:
        try:
            comm.sendRadioMessage(nodeid,
                40000000, msg)
            nodeid = nodeid +1
        except:
            print "\nCommunication failed!!"
# Run in RFM mode for 10 Secs
    comm.waitFor(10 * 4000000)
# Pause simulation to node Power values
# sim.pause() -- Screen snapshot code
```

postcmd "End of 2nd Simulation"

numsec 65
nummotes 5
precmd echo "Running a grid+ random set of 4 motes"
layout gridrand
export DBG=usr2
executable build/pc/main.exe -p
logfile log-gridrand.txt
postcmd "End of 3rd simulation"
Table A.4 (continued)

below works !
    temp = 'import -window root
./LossExptsWithPower/Powerrfm_ '+str(spacing)+'.png'
    put, get = os.popen4(temp)

# Wait till operation is done and Power values recorded
    comm.waitFor(10 * 4000000)
    temp = 'import -window root
./LossExptsWithPower/Powerled_ '+str(spacing)+'.png'
    put, get = os.popen4(temp)

    sim.stop()
    sim.reset()

while start <22:
    start=start+2
    repeat_sim(start, 100)

#sim.reset()
#sim.stop()
sim.exit()