Efficient Over-the-air Remote Reprogramming of Wireless Sensor Networks

by

Nasif Bin Shafi

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
November 2011

Copyright © Nasif Bin Shafi, 2011
Abstract

Over-the-air reprogramming is an important aspect of managing large wireless sensor networks. However, reprogramming deployed sensor networks poses significant challenges due to the energy, processing power and memory limitation of sensor nodes. For improved energy efficiency, a reprogramming mechanism should use less transmission and flash writing overhead. Past research has proposed different mechanisms for reprogramming deployed sensor networks. However, all of these mechanisms produce large patches if software modifications involve changing program layouts and shifting global variables. In addition, existing mechanisms use large amounts of external flash and rewrite entire internal flash. In this thesis, we present a differential reprogramming mechanism called QDiff that mitigates the effects of program layout modifications and retains maximum similarity between old and new software using a clone detection mechanism. Moreover, QDiff organizes the global variables in a novel way that eliminates the effect of variable shifting. Our experiments show that QDiff requires near-zero external flash, and significantly lower internal flash rewriting and transmission overhead than leading existing differential reprogramming mechanisms.
Acknowledgments

First and foremost, I offer my sincerest gratitude to my supervisor, Dr. Hossam Hassenein, who has supported me throughout my thesis with his patience and knowledge, whilst allowing me the room to work in my own way. He has been my inspiration as I hurdle all the obstacles in the completion of this research work. His in-depth knowledge and keen interest in this field, his logical way of thinking, prudent guidance and suggestions at all stages of the work have been very helpful to complete this thesis.

Throughout my research work, I have been blessed with a great colleague Dr. Kashif Ali, who introduced the field of wireless sensor networks, and taught me how to carry on a research work. His extraordinary patience in reading numerous inferior drafts and for making them publishable is really appreciable. Both the syntactical and technical aspects of the text have been improved through his careful reading of the manuscript. I am deeply grateful to him for his patience, keen judgement, valuable advice and cooperation to successfully complete this thesis.

I would like to acknowledge with sincere thanks the all-out cooperation and services rendered by Basia Palmer and other members of TRL lab for many things.
I would like to thank my wife, Safia Hasan and my parents for their constant support, patience and inspiration.

Above all, I would like to thank the omnipresent Allah, for answering my prayers for giving me the strength to finish this thesis successfully.
### List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(C)ELF</td>
<td>(Compact) Executable and Linkable Format</td>
</tr>
<tr>
<td>EEPROM</td>
<td>Electrically Erasable Programmable Read Only Memory</td>
</tr>
<tr>
<td>iHEX</td>
<td>Intel Hexadecimal format</td>
</tr>
<tr>
<td>IoT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LCS</td>
<td>Longest Common Sub-sequence</td>
</tr>
<tr>
<td>LED</td>
<td>Light-emitting Diode</td>
</tr>
<tr>
<td>LiteOS</td>
<td>Lite Operating System</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>nesC</td>
<td>networked embedded systems C</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>ROM</td>
<td>Read-only Memory</td>
</tr>
<tr>
<td>SOS</td>
<td>Sensor Operating System</td>
</tr>
<tr>
<td>SPI</td>
<td>Serial Peripheral Interface</td>
</tr>
<tr>
<td>Symtab</td>
<td>Symbol table</td>
</tr>
<tr>
<td>TinyOS</td>
<td>Tiny Operating System</td>
</tr>
<tr>
<td>WSN</td>
<td>Wireless Sensor Networks</td>
</tr>
</tbody>
</table>
# Contents

Abstract i

Acknowledgments ii

List of Acronyms iv

Contents v

List of Tables vii

List of Figures viii

Chapter 1: Introduction 1
  1.1 Motivation ............................................. 3
  1.2 Thesis Contribution .................................... 5
  1.3 Organization .......................................... 6

Chapter 2: Background and Literature Survey 7
  2.1 Background ............................................. 7
  2.2 Literature Survey ....................................... 14
    2.2.1 Dissemination Protocols ......................... 14
    2.2.2 System Level Reprogramming .................... 16
    2.2.3 Virtual Machine Based Reprogramming .......... 16
    2.2.4 Modular Reprogramming .......................... 17
    2.2.5 Differential Reprogramming .................... 19

Chapter 3: QDiff: A Differential Reprogramming Scheme 22
  3.1 Introduction .......................................... 22
  3.2 Motivation, Problem and Objective .................... 23
  3.3 QDiff: Differential-based Reprogramming Scheme Using Clone Detection 26
  3.4 QDiff Patch Creation Process ......................... 30
    3.4.1 Branches .......................................... 31
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2 Global Variables</td>
<td>38</td>
</tr>
<tr>
<td>3.4.3 Indirect Addressing</td>
<td>45</td>
</tr>
<tr>
<td>3.4.4 Relative Jumps</td>
<td>52</td>
</tr>
<tr>
<td>3.5 QDiff Patch Deployment Process</td>
<td>53</td>
</tr>
<tr>
<td>Chapter 4: Implementation and Evaluation</td>
<td></td>
</tr>
<tr>
<td>4.1 Implementation details</td>
<td>57</td>
</tr>
<tr>
<td>4.2 Evaluation Scenarios</td>
<td>59</td>
</tr>
<tr>
<td>4.3 Quantitative Performance Metrics</td>
<td>71</td>
</tr>
<tr>
<td>4.4 Performance Evaluation</td>
<td></td>
</tr>
<tr>
<td>4.4.1 Patch size</td>
<td>72</td>
</tr>
<tr>
<td>4.4.2 Internal flash writing</td>
<td>73</td>
</tr>
<tr>
<td>4.4.3 External flash usage</td>
<td>75</td>
</tr>
<tr>
<td>4.4.4 System Restart</td>
<td>75</td>
</tr>
<tr>
<td>4.4.5 Code Overheads</td>
<td>77</td>
</tr>
<tr>
<td>4.4.6 Heterogeneity</td>
<td>78</td>
</tr>
<tr>
<td>Chapter 5: Conclusion</td>
<td>79</td>
</tr>
<tr>
<td>Bibliography</td>
<td>81</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Internal flash memory bytes comparison for the Stream, Hermes and QDiff ........................................... 75

4.2 External flash memory bytes comparison for the Stream, Hermes and QDiff ........................................... 76

4.3 Qualitative comparison of various reprogramming schemes ............................................................... 78
List of Figures

1.1 Generic architecture of WSNs mote ............................... 2
1.2 Software architecture of sensor mote(modified from [56]) .......... 3
2.1 Operating system architecture for wireless sensor networks ......... 8
2.2 Virtual machine with other layers ..................................... 10
2.3 Layout of an Executable and Linkable Format (ELF) file .......... 13
2.4 Classification of over-the-air reprogramming in WSNs .............. 15
3.1 Important steps of a differential-based reprogramming scheme .... 27
3.2 Major steps of the QDiff differential reprogramming scheme ....... 28
3.3 Example of function call address change due to code insertion .... 32
3.4 Example of address change due to function reordering .............. 32
3.5 Padding newly added code block at the end of program code ....... 33
3.6 An illustration of QDiff scheme, handling simultaneous function insertion and deletion .................................................. 35
3.7 Flowchart for fixing branches ........................................... 36
3.8 An exemplary scenario showing addition of multiple functions .... 37
3.9 Illustration of QDiff handling multiple functions .................... 38
3.10 Comparison of bss section for traditional and QDiff ELF file .... 39
3.11 Illustration of traditional ELF and QDiff modified data/bss sections 41
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.12</td>
<td>Illustration of global variable addition, both initialized and uninitialized.</td>
<td>42</td>
</tr>
<tr>
<td>3.13</td>
<td>Sequence of variables including array after addition in bss section of GCC compiled ELF file.</td>
<td>44</td>
</tr>
<tr>
<td>3.14</td>
<td>Sequence of variables including array in bss section of QDiff modified ELF file.</td>
<td>45</td>
</tr>
<tr>
<td>3.15</td>
<td>Flowchart for fixing global variables.</td>
<td>46</td>
</tr>
<tr>
<td>3.16</td>
<td>Illustration of indirect address access and global variables, for traditional and modified ELF files.</td>
<td>48</td>
</tr>
<tr>
<td>3.17</td>
<td>Illustration of indirect address access and global arrays, for traditional and modified ELF files.</td>
<td>49</td>
</tr>
<tr>
<td>3.18</td>
<td>Flowchart for fixing indirect addresses.</td>
<td>51</td>
</tr>
<tr>
<td>3.19</td>
<td>Illustration of change in the relative jump instructions.</td>
<td>53</td>
</tr>
<tr>
<td>3.20</td>
<td>Illustration of QDiff changes to handle relative jumps.</td>
<td>54</td>
</tr>
<tr>
<td>3.21</td>
<td>Flowchart for fixing relative jumps.</td>
<td>55</td>
</tr>
<tr>
<td>3.22</td>
<td>Flowchart for QDiff.</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>An IRIS sensor mote(LEDs are shown in circle)</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Case 1: Timer value is changed.</td>
<td>63</td>
</tr>
<tr>
<td>4.3</td>
<td>Case 2: Addition of one toggle in Timer0.</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Case 3: Addition of one toggle in each timer i.e. Timer0 toggles LED0 and LED1, Timer1 toggles LED1 and LED2, Timer2 toggles LED2 and LED0.</td>
<td>65</td>
</tr>
<tr>
<td>4.5</td>
<td>Case 4: Toggles are reorganized i.e. Timer0 toggles LED2 and LED0, Timer1 toggles LED2 and LED1, Timer2 toggles LED1 and LED0.</td>
<td>66</td>
</tr>
</tbody>
</table>
4.6 Case 5: RadioCountToLeds is modified to display last two bits on its LEDs and value displayed in third LED is discarded. 67

4.7 Case 6: RadioCountToLeds is modified to display 1’s complement of least significant three bits of received counter. 68

4.8 Case 7: RadioCountToLeds is modified to display most significant three bits of received counter value. 69

4.9 Case 8: RadioCountToLeds is modified to display counters bits if it has an even value, otherwise no change is made in sensor mote LEDs. 70

4.10 Patch size comparison for the Stream, Hermes and QDiff reprogramming schemes. 73

4.11 Internal flash writing comparison for the Stream, Hermes and QDiff reprogramming schemes. 74

4.12 External flash memory pages comparison for the Stream, Hermes and QDiff reprogramming schemes. 76
Chapter 1

Introduction

Wireless Sensor Networks (WSNs) is attracting great deal of attention from both academic and industrial researchers. A typical network is composed of a large number of small-sized, battery-powered or energy harvesting wireless sensor nodes that are limited in available power, processing, and communication capability. Wireless sensor networks have a broad range of applications that include battlefield surveillance [4,24], industrial process monitoring and control [43,68], Habitat monitoring [44], Volcano Monitoring [64], wildlife monitoring [31], structure monitoring [33,66], Health care monitoring [8,30,36], underground mining [29,41,63], etc. Recently, the wireless sensor networks is being evaluated as the main enabling technology behind the highly anticipated Internet of Things (IoT) [22].

A node, commonly known as mote, of a typical sensor network consists of diverse hardware components, each with distinct capabilities and functionalities. Typical components include radio transceiver, sensor(s) (temperature, pressure, humidity, etc.), actuator(s), processing unit, memory storage, etc. These components are low-end units with limited capacity and functionalities. For instance, a typical Crossbow mote memory is limited to 128-256 KB ROM and 4-8 KB RAM [53,65], with limited
availability of external flash memory. Similarly the processing capability and available energy is scarce and limited to few MHz and joules, respectively. A generic mote architecture is depicted in Fig. 1.1.

The overall software stack of a WSN mote is composed of three layers. First, the bottom-most layer, is the operating system. The scope of operating system is confined to each mote as it provides an interface between hardware (device drivers) and application layers. Second, the middle layer, glues the operating system provided abstract with the application layer. Third, the top layer, is the WSN application layer wherein the network application resides. The scope of middleware and application layers is the whole sensor network. Fig.1.2 shows the overall software architecture.

Development and deployment of the WSNs is tightly coupled with the desired application(s). The nature of application development requires frequent updates (re-programming) of its software components. These updates can be small fixes, few lines only, e.g. bug fixes, or bigger, e.g. frequent code updates, new components and new applications. Unlike conventional networks, WSNs are deployed in diverse environments therefore manual collection and update of motes is both costly and
1.1. MOTIVATION

Unlike traditional network nodes, reprogramming of the software components of motes is more challenging because of the limited power, processing, storage and communication capabilities. Significant research is done in order to address the resource-challenged nature of WSNs.

TinyOS [38], the de-facto and dominating monolithic operating system in WSNs,
provides native reprogramming facilities (deluge [26]). The deluge, mainly to conserve memory, is tightly coupled with a TinyOS application. However, due to tight coupling, a minor software update requires complete retransmission and reprogramming of the mote. For instance, the simple Blink application (periodic blinking of an LED) without and with deluge is 2128 bytes and 35 kilobytes, respectively. Any modification would require the base-station to transmit the new Blink with deluge, i.e. 35 kilobytes, for every node in the network. This consumes significant communication, processing and storage resources.

An alternate solution is modularization of the mote’s operating system architecture. This results in transmission of only modified modules which then are linked and loaded by the OS. However, such feature is only available in modular operating systems. Currently, operating system modularity in WSNs is supported by very few systems, SOS [23], LiteOS [6] and RETOS [7]. However, these operating systems have limited visibility. Contiki [18], an emerging alternate to TinyOS, supports modularity but at cost of extra flash storage and processing. The extra storage is for keeping track of the symbol table. Processing overhead arises from various book-keeping tasks to resolve cross referenced symbols that are required for linking and loading of new module. Furthermore, the granularity of any software update is still confined to transmission and dynamic linking and loading of the updated module.

Any existing reprogramming approach requires full flash memory rewriting and consumes significantly higher power and time. Flash memory writing requires a specific voltage, higher than the microcontroller required voltage, mainly to ensure safe memory writing. Furthermore, these schemes consume large portions of available external memory to store the golden image and associated patches. Such storage
1.2. THESIS CONTRIBUTION

overlaps and limits other critical operation and functional capacity of the network, e.g. store and forward routing [12,20,45,52], event logging [10,11,54] and real-time debugging support [9]. In addition, writing to an external memory is slow and requires significantly more power than writing to other kinds of memory. Any additional external memory solely for reprogramming would result in increased mote size, cost and power requirements.

Although some existing over-the-air reprogramming schemes require less data to transmit, they may actually increase the size of program’s code. Furthermore, they may also require a change in the compiler. For instance, Elon [14], requires to change the networked embedded systems C(nesC) [21] compiler. Other reprogramming schemes, introduce complex bootloader and extra jump tables [49,51], tradeoffs such as large volume of stored meta data for smaller patches [13] and code fragmentation [34]. Furthermore, all of the existing reprogramming schemes perform extremely poor and produce large patches, if function or variable layouts are changed.

1.2 Thesis Contribution

Our research aims primarily at devising an efficient over-the-air reprogramming scheme for WSNs to produce small size patches without the need of whole flash memory rewriting, large amount of external memory storage, complex bootloader and compiler changes. The objective is to provide remote over-the-air reprogramming of WSNs motes while reducing the complexity and energy consumption, to prolong the lifetime and operational capacity of network.

The main contribution of this thesis is devising and implementing an energy-efficient and low-complexity over-the-air reprogramming scheme, namely QDiff (Queen’s
DIFFerential). The proposed QDiff scheme calculates the minimum size patch by keeping maximum similarity between two programs, old and new, using clone detection. Clone detection is used to find similar functions and variables hence, to re-organize function and variable layout to keep the maximum similarity between old and new programs. New functions or variables are padded at the end of the program. Most of the existing reprogramming mechanisms fail to handle global variable shifting due to insertion or deletion of variables. QDiff proposes a new re-organization way of global variables which efficiently eliminates the problem of global variable shifting. These two approaches drastically reduce the size of patch, while facilitating maximum possible function and variable layout changes.

To evaluate the performance, we have implemented QDiff in TinyOS [38] using an IRIS [65] mote platform and compared with prominent over-the-air reprogramming schemes for WSNs. As comparison metrics we use patch size, frequency of flash writings, size of external memory, and consumed time and energy. We also investigate the overheads of the QDiff approach of reprogramming.

1.3 Organization

The remainder of this document is organized as follows. Chapter 2 presents background and literature survey of reprogramming in WSNs. Chapter 3 introduces the QDiff scheme and presents the working details. Chapter 4 presents implementation details, evaluation methodology and experimental results. Finally, Chapter 5 concludes this document by highlighting the main issues addressed in the thesis and outlining some of the future research directions.
Chapter 2

Background and Literature Survey

2.1 Background

In this section, we briefly present some basic definitions and explain over-the-air reprogramming in wireless sensor networks.

Monolithic System: In a monolithic system, OS kernel and other components are joined together in a single system image. No barrier or privilege restrictions exist amongst the modules, i.e. each module can directly invoke any other module [59]. Typically, a monolithic system follows static design approach i.e. all system parameters should be specified at design time. Monolithic architecture is suitable where system updates are less frequent. Fig. 2.1-a shows the monolithic architecture.

Modular System: In a modular system, the OS is split amongst different interacting modules with minimum possible kernel size. The partition between kernel and other OS parts is performed at design time. The kernel provides with some basic
Figure 2.1: Operating system architecture for wireless sensor networks.
and mandatory functionalities e.g. scheduler, dynamic loading or unloading of modules, hardware access interface, communication stack, and so on. Other common OS functionalities, e.g. power management, file system, advanced network protocol, etc. are implemented as external library and modules. These modules and libraries are loaded and linked by the kernel on need basis. Interaction between different modules are supported via indirection table provided by jump table and maintained by the kernel. Fig. 2.1-b shows the modular architecture.

**Virtual Machine:** It is a software implementation of a machine (i.e. a mote) which executes programs like a physical machine and in complete isolation of the host OS. Virtual machines can either be a single process, group of processes or whole operating system. The prominent example of virtual machines is Java Virtual Machine (JVM). JVM runs over the native machine as it executes the java bytecodes. Similarly, the virtual machine in WSNs runs over the native OS of the mote while executing the bytecodes. The bytecodes are instruction sets which have one-byte opcodes followed by optional parameters. Bytecodes may often be directly executed on a virtual machine (i.e. interpreter), or it may be further compiled into machine code for better performance. An overview of virtual machine layers, in context of others, is shown in Fig. 2.2.

**Symbol Table:** The compiler keeps track of all identifiers, e.g. variables, labels, functions and their properties e.g. value, type, scope, address, etc. in the symbol table. It plays an important role in linking of object files into a single executable file. The symbol table is also used for re-engineering and debugging purposes.
2.1. BACKGROUND

Figure 2.2: Virtual machine with other layers.

**Linking and Loading:** Linking means creation of a single executable file from multiple object files. The compiler parses each file and creates intermediate object files. Each object file has its own symbol and relocation table. The linker uses these tables to resolve identifier addresses, values and scope in order to create an executable file. Loading means the procedure of loading the executable file at some memory address for processing unit to start execution.

**Pre-linking:** Dynamic linking and loading results in significant performance degradation of system. To improve system performance, the kernel caches previous linking and loading information, hence significantly reducing the linking and loading time. To maintain system integrity, before re-loading, the system needs to be notified of any updates.
2.1. BACKGROUND

**Dynamic Linking and Loading:** Dynamic linking and loading facilitates on-demand loading of various OS modules, hence reducing memory requirements. In dynamic linking and loading, each module maintains its own symbol table which is used to resolve any cross references. In WSNs, dynamic symbol resolution results in overall system degradation as, for every resolution, it may require reading certain modules from external flash memory. The kernel reads the module from external flash using peripheral interfaces, generally through Serial Peripheral Interface (SPI). Moving data between external and internal flash memory is both time and energy consuming. Furthermore, unloading of an unused module introduces extra overhead.

**Initializer:** The initializer describes the entry point from where system starts the execution of the program code. Functional requirements of an initializer is highly system dependent. In the context of WSNs, an initializer performs the following main tasks: load global variables to RAM, initialize the interrupt vector table, initialize devices in sensor mote and start the main routine’s execution from a pre-determined entry point.

**Executable and Linkable Format (ELF):** The Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps. An ELF file contains many segments, as shown in Fig. 2.3 and are explained below.

- Program Header: The program header contains generic information for the rest of the ELF file.

- Section Header: The section header contains the information of a particular
section. The section includes code section, data section and symbol table section.

- rodata: This section contains constant string, e.g. used by printf statements.
- reltext: This section contains relocatable code.
- .Code: This section contains the machine code, i.e. the executables.
- .BSS: This section contains the uninitialized global variables.
- .Data: This section contains the initialized global variables.
- .symtab: This section contains symbol table, list of different identifiers, and their properties, i.e. address, value, and scope.

**Watchdog Timer:** The watchdog timer is a fail-safe mechanism in embedded systems. The watchdog timer, in case the system goes into an undefined state, is used to restore (reset) the system to an acceptable state.

**Golden Image:** The golden image is the latest software version that ran successfully on the mote. In case of any problem, e.g. software bug, the watchdog timer is used to flush out existing image and replace it with the golden version.

**Clone Detection:** Clone detection is a software re-engineering approach to detect code similarity between two versions. Generally, clones are classified into following four classes:

- Type 1 clone: Two codes are almost similar except for some differences in white spaces and comments.
Figure 2.3: Layout of an Executable and Linkable Format (ELF) file
2.2. LITERATURE SURVEY

- Type 2 clone: Identifier names, e.g. variables and functions have been changed along with white spaces and comments.
- Type 3 clone: Addition (or deletion) of one or more lines of code or functions.
- Type 4 clone: Significant code change with minor structural similarity.

2.2 Literature Survey

Wireless reprogramming is an active research area and can be classified into dissemination protocols and reprogramming schemes. Dissemination protocols are used for efficient and low-power transfer of patch updates to motes within the deployed WSNs. Reprogramming schemes can further be classified as system level, module level, virtual machine and differential approaches. The overall classification of wireless reprogramming is shown in Fig. 2.4. In rest of this section, we survey the prominent dissemination protocols and over-the-air reprogramming schemes.

2.2.1 Dissemination Protocols

XNP [27] is a single-hop reprogramming protocol. In XNP, upon receiving the patch packets, the sensor mote re-broadcasts to its own neighbours. Any missing or corrupted packet is requested directly from the base-station. Due to the broadcast nature of the protocol, XNP has significant communication overheads. Trickle [39] is the first multi-hop code dissemination protocol for reprogramming WSNs. Although it alleviates the broadcasting problem, it can only transmit small size patches.

Deluge [26] is an extension of the Trickle protocol. Deluge is a well known and widely used multi-hop code dissemination protocol for bulk data transfer. Deluge
2.2. LITERATURE SURVEY

Figure 2.4: Classification of over-the-air reprogramming in WSNs.

reduces transmission time by using pipelined data transfer. Deluge transfers the whole software image while reprogramming. Moreover, Deluge is tightly coupled with the software of the sensor mote in the final executable file. Due to this tight coupling, binary code of Deluge is indirectly transmitted over the network in every software update. Therefore, overall data transmission is very large, which results in a large number of energy hungry flash writings.

Stream [50] improves on the shortcomings of the Deluge protocol. Stream isolates the reprogramming protocol from the application hence, overcomes the problem of sending reprogramming protocol with every update. To enable this, the reprogramming protocol is consistently stored in mote’s external memory (if available). The
Stream protocol requires complete transmission of application even for small changes.

### 2.2.2 System Level Reprogramming

In system-level reprogramming, the existing software is completely replaced by an updated system. The updated system, stored in external flash memory, is copied into internal flash memory upon system restart. System-level reprogramming is highly inefficient as a very small update in the existing system requires replacing the whole existing system. Existing monolithic operating systems, e.g. TinyOS and Nano-RK [19], due to their single image file, can make use of the system level reprogramming technique. Certain non-monolithic operating systems, e.g. Mantis [3], also depend on system-level reprogramming due to the lack of support of more advanced reprogramming techniques. Even for minor code changes, system-level reprogramming requires whole image transmission, requires more power and time and has significantly higher flash memory writings. Furthermore, the mote, to store the complete system image, requires additional external flash memory. The system-level reprogramming approach however, is simple and provides better reliability. Also due to complete flushing, system developers need not to be worried about illegal data-access, illegal jumps and complex bootloaders.

### 2.2.3 Virtual Machine Based Reprogramming

In virtual machine based WSNs, each mote runs an instance of the virtual machine. The virtual machine is used for execution of both on-network packets and bytecode instructions. Mate [40] is the first virtual machine in WSNs built on TinyOS. Mate receives instructions in the form of network packets called capsules. The capsule, 23
bytes in length, contains any code update or patches. Another WSNs virtual machine
namely, VM* [35] adopts the JVM architecture. More in-depth technical details of
WSNs virtual machines can be found in [5, 46, 47, 58].

Minimum data transfer is required to update an application running on a vir-
tual machine. However, the nature of virtual machines demand significantly high
processing time and capabilities to be provided by the mote’s CPU. Furthermore,
the reprogramming scope of virtual machine is limited only to very few high-level
instructions.

2.2.4 Modular Reprogramming

Contiki is the first WSNs operating system to support modular update and consists
of two main components: system core and loaded program. The Contiki Core, with
exception of the bootloader, is a non-reprogrammable component [18]. Therefore,
any code change to the kernel, program loader, symbol table and communication
interfaces is not supported. However, enhanced functionalities, e.g. file system sup-
port, shell support, power management are loaded modules and are reprogrammable.
The Contiki follows a customized version of ELF format [37], known as Compact
ELF(CELF) [16]. The CELF file contains all relocation information which is used
by the program loader to obtain information about the module memory size require-
ments and to load it into code flash. Furthermore, the relocation information is used
to update the symbol tables. For greater flexibility, these modules are designed in a
loosely-coupled manner and communicate via the kernel. Although this introduces an
extra level of indirection, it allow more flexiblility in modular reprogramming. Contiki
also introduces dynamic linking and loading, an important feature to support modular
reprogramming. In [61], GZIP is proposed, which is a zip and unzipping mechanism specially designed for code transfer. As dynamic linking and loading causes some performance degradation, the authors of reference [15] propose to perform linking and loading by using pre-linking data. Usage of pre-linking data saves large amount of time and energy. Furthermore, the modular approach adopted by Contiki results in larger CLEF files, maintaining additional symbol table(s) and modules security concerns.

Similar to Contiki, SOS [23] also adopts the modular approach. In SOS, each module has a well defined entry and exit point and designed in a loosely coupled manner. Interactions among modules are accomplished via message passing, direct calling of registered functions of modules or kernel system calls. All modules implement functions for two common messages, “init” and “final”. Functions for “init” and “final” set the module’s initial state and release all resources acquired by module, respectively. Messaging is highly flexible and asynchronous and provides greater independence among modules. But, this flexibility and independence come with slow response. Sometimes synchronous action from different modules is required to perform certain tasks. Message passing is not a suitable option in this case. To solve this issue, SOS allows modules to register some function of other modules. A module can call these registered functions directly. Modules have to access the kernel via a jump table which introduces additional indirection.

Modules in SOS use position independent code and provide more flexible and robust way of interaction among different modules. Position independent code [37] can be properly executed regardless of the memory position. Executable files with relocation information are larger in size compared to the executable files with position
2.2. LITERATURE SURVEY

independent code as position independent code requires no relocation information.

LiteOS [6] and RETOS [7] use the Contiki memory relocation approach, however unlike Contiki, LiteOS uses Intel HE(X)(iHE) format [37] to store relocation information. HEX files require lower memory size than the ELF file. Furthermore, LiteOS provides Unix-like abstractions, hence, updating a module is highly user friendly. User modules can access a system resources through a special type of function pointer *callgate* [6]. RETOS uses non-standard file system for storing relocation information. Other operating systems, e.g. CORMOS [67], AmbientRT [25] also follow a dynamic and modular architecture. Dynamic TinyOS [48] attempts to achieve modular architecture within TinyOS.

The modular architecture is suitable for over-the-air reprogramming. Unlike the monolithic architecture, any system change is local, only the updated modules need to be transmitted. However, a large-memory footprint and slow system execution are inherent disadvantage of any modular system. Reliability is also another issue as buggy or poorly tested module might cause the whole network to crash. A possible approach to security, reliability and undefined system behaviour is using frequent restarts using a watchdog timer at the cost of higher power consumption and data loss for the motes.

2.2.5 Differential Reprogramming

In the differential reprogramming scheme, the base-station generates a patch using the difference between the old and updated program. Rsync [60] is a well known differential update scheme that is widely used in desktops and servers. The Rsync divides the program into different blocks and calculates the hash value of those blocks.
The hash values are then matched to determine the block’s insertion, deletion, or modification. In reference [28], the authors present a rsync-based scheme to generate small sized patch. However, this scheme does not produce an optimal result as it discards ELF the file structure while calculating diff. It requires rewriting the whole flash memory as well as a large amount of external flash memory. Moreover, the scheme is not optimized for call, jump and function rearrangement.

In reference [34], the authors propose a differential scheme, which changes the linking procedure and keeps a slop\textsuperscript{1} region after each function to reduce the patch size caused by call and jump instruction. This scheme creates a small patch, however it causes fragmentation of flash memory. Moreover, since the size of the slop space is heuristically assigned, which leads to inefficient utilization of flash memory.

Zephyr [51] keeps a jump table via which all call and jump go to their destination. This mechanism reduces patch size along with mote’s energy and processing requirements. However, the scheme shows poor performance with applications with loops, which is common in WSNs. Hermes [49], an improvement to Zephyr, removes the slop region for global variables and jump table at the cost of a complex bootloader. A complex bootloader requires significantly larger boot memory. Moreover, zephyr and Hermes both require complete flash memory rewriting and a large amount of external memory.

Elon [14] overcomes the restart requirement of earlier schemes while also reducing flash memory access. Elon divides the system into two different components namely, replaceable and non-replaceable. The core of the operating system is considered non-replaceable and other parts are considered replaceable components. Although Elon does not require mote restart, it is highly platform specific. For instance, currently the

\textsuperscript{1}Slop region means free space
scheme, by design, can only work on a TelosB mote running TinyOS. Furthermore, Elon also requires significant changes in the compiler to support new programming language syntax.

$R^2$ [13] reduces the patch size by using an efficient implementation of dynamic loading and linking modules. The $R^2$ scheme maintains meta-data regrading changeable information and the difference of this meta-data is transmitted to motes. Using meta-data eliminates the need of a jump table, which is a typical requirement of Hermes or Zephyr, and complex bootloaders. However, this scheme requires full flash memory writing. Moreover, a significantly large amount of code flash memory is required to maintain the meta data.

All of the aforementioned schemes perform poorly when there is a change of program and variable layout, require full flash memory writing and large amount of external flash memory. Our proposed scheme avoids such difficulties by re-organizing global variables, figuring out the difference using clone detection and maintaining maximum similarity between two versions of software.
Chapter 3

QDiff: A Differential Reprogramming Scheme

3.1 Introduction

The focus of our work is to propose an efficient reprogramming scheme for WSNs. The efficiency is defined in terms of small patch size, near zero external flash reads/writes, minimum internal flash reads/writes and the elimination of a system restart after applying the patch. We call our scheme Queen’s DIFFerential (QDiff) reprogramming. QDiff is a differential-based reprogramming scheme, which utilizes clone detection to determine code changes efficiently. Furthermore, QDiff handles branches, global variables, indirect addresses and relative branches by amending the ELF format in a manner which is compatible with standard ELF. These effective amendments dramatically reduce the internal flash memory usage and require near zero external flash memory. Furthermore, the wireless sensor motes require no restart after applying the QDiff generated patch.

The remainder of this chapter is organized as follows. Section 3.2 outlines the problems and objectives of the differential reprogramming scheme. Section 3.3 explains the proposed scheme (QDiff). Section 3.4 describes the patch creation process.
and using examples, explains the QDiff approach to handle changes in program line of code, functions, branches, global variables, indirect addresses, and relative branches. Section 3.5 explains the QDiff approach of applying the generated patch.

3.2 Motivation, Problem and Objective

Motivation:

Traditional differential utilities, e.g. Unix diff, kdiff are based on the Longest Common Sub-sequence(LCS) algorithm, that calculates a “delta” between two files without structural consideration. For instance, simply moving a function location within the code, results in a large patch. Another well known differential utility is rsync. Rsync is based hash value calculation. In rsync, two programs “old” and “new”, are divided into blocks and a hash value is calculated for each block. By comparing these hash values, a patch is generated for the updated code. Rsync-based differential approaches are considerably faster than the LCS-based approaches. However, they are less accurate.

Traditional differential approaches are designed for desktops and servers, wherein having a patch of few kilobytes (or even megabytes) in size is the norm. However, this is a serious issue for the motes in the wireless sensor networks, due to their constrained resources, functional limitations and unconventional tradeoffs. For instance, mote designers rely more on computation than transmission, since transmission of one byte of data is more costly, in terms of energy, than executing thousands of instructions [32]. Similarly, flash memory read and write, both internal and external, consume significant power. Moreover, the flash memory writing is a block based mechanism, i.e. even if a single byte is changed, the whole block must be rewritten.
Furthermore, flash memory writing requires higher (2.7V) than minimal operational voltage (1.7V), at which its behaviour is unpredictable. In GreenOrbs [42] WSNs application, researchers found that for the TelosB [53] WSN mote, the network lifetime is reduced from 40 days to 11 days upon using a reprogramming protocol with full flash memory writing [14]. Therefore, reducing flash writing reduces the energy consumption of the mote, hence allowing more of the mote reprogramming while maintaining the network lifetime. A large number of reprogramming schemes have been proposed in the literature [49,51,57], however all of them require flash rewriting and system restart. To the best of our knowledge, our proposed approach is the first scheme which produces such a small patch, does not depend on flash rewritings and does not require mote restart.

Problems:
A single line of code change, in a high-level programming language, may cause a significantly larger change at the lower-level, e.g. assembly code. To address this, existing solutions [14, 34, 51] require changes in the programming language syntax or the compiler itself and hence applies only for certain platforms. A heterogenous solution, on the other hand, although independent of compiler and mote architecture, needs to solve the following technical issues:

- **Functions:** The compiler might change its functional layout for optimization purposes resulting in a large patch file. Moreover, these changes shift the machine code in the final ELF file. Furthermore, even a small change, e.g. insertion of single instruction, may shift all of its subsequent instructions and branch addresses. This will yield significantly large patch file whose deployment would
require full rewriting of internal flash.

- **Global variables:** During the creation of an executable file, the compiler considers the global variables in two contexts: initialized and uninitialized. Initialized global variables are kept in the data section, and uninitialized global variables are kept in the bss\(^1\) section. To optimize memory space, both of these sections (data and bss) are kept back-to-back without any slots in the memory space. Due to this, any addition in data section, e.g. a new variable, shifts the address of all variables within the bss section. Furthermore, the shift also modifies all instructions that reference the shifted variables, hence results in a very large patch file. Moreover, there is no assurance that a compiler will keep the same variable layout every time the program is compiled.

- **Relative jumps:** The relative jump range is calculated using the current memory location. Based on current memory address a relative jump can either be a positive offset or a negative offset. Positive offset means that the jump target address is higher than the current address. Negative offset means that the jump target address is lower than the current address. Addition or removal of any new or existing instructions, between the current memory location and the target location, will result a change in the jump offsets. This happens even if the instructions at the target location has not been modified, hence resulting in unnecessary large patch file.

- **Indirect addresses:** Indirect addressing is an important part of the load-store architecture of the Reduced Instruction Set Computing (RISC) machine. The

\(^1\)Bss section is a part of ELF file. Historically, BSS(Block Started by Symbol) was a pseudo-operation in an assembler, developed for the IBM 704.
RISC architecture does not allow to direct access to the memory location. Memory locations are accessed or modified using registers. Indirect addressing provides fast access to large data structures, e.g. arrays, linked lists, union, etc. However, any changes in the global variable layout will also change the corresponding indirect instructions and must be taken care of in order to produce correct patch files.

**Design Goals:**

The main objectives of the differential-based reprogramming scheme are as follows:

- Minimizing difference size, as small difference means less energy and time to reprogram the network.
- Minimizing flash memory writing, as flash memory writing is energy hungry and reduction of flash memory writing means better energy and time efficiency.
- Minimizing external flash requirement, as reducing external flash usage means reducing cost, as well as reducing requirement for board real estate.
- Supporting heterogeneity as many different hardware platforms and operating systems are available for wireless sensor networks, with diverse specifications.
- Eliminating the need for mote reboot after applying the patch on mote software.

### 3.3 QDiff: Differential-based Reprogramming Scheme Using Clone Detection

A differential-based reprogramming scheme consists of several major steps. These are, receive old and new files from code database, calculate the patch using an “algorithm”, compress and encode patch, store (old, new, patch) tuple in the database
3.3. QDIFF: DIFFERENTIAL-BASED REPROGRAMMING SCHEME USING CLONE DETECTION

Figure 3.1: Important steps of a differential-based reprogramming scheme.

(for future usages), transfer the encoded patch over the network, sensor mote receives patch and stores it in memory, bootloader reads, parses, and applies that patch on existing firmware and finally, re-start the application. Figure 3.1 illustrates the steps of differential reprogramming.

The focus of our research is to devise an efficient differential scheme to fulfill the aforementioned objectives (shown in bold in Fig. 3.1). For this purpose, we propose a novel differential scheme, namely QDiff, with the objectives described above. This approach provides significant advantages over other schemes [14,17,49].

To calculate the patch file, QDiff considers both executable files, i.e. the ELF file, as well as the high-level source code. Making use of high-level programming languages provides significant advantage over other schemes, which only use low-level, because its grammatical and syntax rules provide better flexibility to determine code similarities. The code similarities are determined using clone detection tools. The reprogramming steps of QDiff are shown in Fig. 3.2 and are explained next.
3.3. QDIFF: DIFFERENTIAL-BASED REPROGRAMMING SCHEME USING CLONE DETECTION

Figure 3.2: Major steps of the QDiff differential reprogramming scheme.
1. QDiff calculates clones between the old and the new file using clone detection tool. The clone detection tool takes in the C file of the old and the modified program and calculates a mapping between functions and variables. We assume that the clone detection mechanism has perfect precision. The C files, in case of TinyOS programming, are generated by the compiler as it converts the nesC into C before it create the final ELF file. The mapping is a list of function and variable clones between the old and the new file, i.e. FunctionClonePair(FunctionOld, FunctionNew) and VariableClonePair(OldVariable, NewVariable). The clone detection algorithm can detect clones under various circumstances, e.g. change in file layout, rename of functions and variables, addition or removal of one or more lines of code, etc. For this thesis, we emphasize “Type 3” clone detection as it can detect addition and removal of new code, functions and variables.

2. QDiff disassembles the ELF files (both old and modified), using the core dump utilities, to determine different sections, i.e. code, data and bss. In this step, mappings between functions and global variables, computed from the clone detection step are used for the function and the variable reordering. This step is further subdivided as follows:

(a) Reorder functions and global variables to enhance the similarity between old and new programs. Place new functions, initialized and uninitialized global variables at the end of the code, data and bss sections respectively.

(b) Change all references to the re-ordered functions and variables. For instance, if a call instruction invokes a function func1 at address 0x234 and
reordering caused the change of \textit{func1} location to 0x456, all call instructions invoking \textit{func1} change their destination address to 0x456. A similar approach is taken for global variables.

(c) Modify the organization of uninitialized global variables, i.e. variables in the bss section. Organize uninitialized global variables in stack fashion. Place new variables at the top of the bss section.

(d) Calculate the delta between the old and the new code section. This includes clone functions and newly added (or removed) functions.

(e) Calculate the delta between the old and the new data section. This includes added, deleted or modified initialized variables.

(f) Calculate delta between old and new bss section. This include added, deleted or modified uninitialized variables.

(g) Aggregate all deltas and calculate the final patch file. The file is optimized to minimize the patch size and flash memory access.

3. The base-station transmits the patch file over wireless medium, which is dispersed through whole network using an existing dissemination protocol.

4. Sensor mote receives the patch file and invokes the bootloader, which applies it without rebooting the mote.

3.4 QDiff Patch Creation Process

Our examination of code changes in low-level language shows that a differential patch creation needs to address four major technical issues: branches, global variables,
relative jumps and indirect addresses. In this section, we explain how QDiff solves these issues.

### 3.4.1 Branches

In this section, we discuss the impact of a code change on branch instructions, i.e. function call and jump to some absolute address. The code change affects branch instructions in two ways. First, an insertion or deletion of new instruction(s) may shifts all the subsequent instructions. This shift changes all call and jump instructions that invoke functions or jump to the location in shifted portion. Fig. 3.3 illustrates change in the functions (funcA and funcB) location due to insertion of the new code, i.e. from 0x3f2 and 0x4d2 to 0x4f2 and 0x5d2 for funcA and funcB, respectively. This change modifies all offsets for call instruction, e.g. from “call 0x3f2” to “call 0x4f2”, for the funcA. Second, the compiler may change the function order during code optimization. This reordering changes functions’ locations and subsequently requires to change all call instructions referring to them. Fig 3.4 illustrates change of function location due to reordering and hence, subsequent modifications of their respective call instructions.

QDiff solves both scenarios by appending any modification to the end of the ELF file. In case of compiler reordering and renaming, the branch related issues are solved using clone detection. If any new function is added, in the middle of code, QDiff moves its location and change all branch instruction that are referring to it. The main idea is to minimize the flash memory writing as only the new functions and the instructions referring it, need to be rewritten. On the other hand, if new instructions are added within the function, QDiff modifies the program to bring maximum similarity in two
Figure 3.3: Example of function call address change due to code insertion.

Figure 3.4: Example of address change due to function reordering.
ways. First, if there is a slop region after that function, the function is extended to that slop region. This slop region has been created due to the deletion of a function. Second, if there is no slop region, new instructions are moved and padded at the end of program. From the callee function, new instructions are added by QDiff; jump to the newly padded instruction and return back to callee function. This is illustrated in Fig. 3.5.

Figure 3.5: Padding newly added code block at the end of program code.
Code deletion, related to branches, comes in two forms: removal of a complete function or some code from within a function. In the first case, if the function is deleted from program, a slop region is inserted in place of the deleted function. The slop regions facilitate future growth of existing functions or insertion of new functions. Moreover, this provides more similarity between old and new program and reduces patch file size. To keep the patch file at minimum the slop regions are kept within or after the function.

In case of simultaneous addition and deletion of function (or code within) is a multi-step process. First, the deleted functions are removed from the old program. This removal creates slop regions within the old program. Second, for the newly added code to function, all matching slop regions are first determined. Matching slop-regions refer those slop-regions, whose size is larger or equal to newly added code/function. To break the tie, QDiff selects the largest region. The slop region is maximally utilized by inserting maximum possible functions. The maximum slop region utilization problem is a variant of a typical (0,1) knapsack problem. If there is no slop region whose size is larger than the function size then the new function is padded at the end. If some code is inserted in an existing function and the existing function has some slop region following it, the function is expanded. Otherwise, new instructions are padded at the end. A call-return pair is inserted into program code to jump to/from the newly padded instructions. An illustration of QDiff approach to handle simultaneous insertion and deletion is shown in Fig. 3.6. The algorithm flowchart is shown in Fig. 3.7.

An exemplary scenario, with addition of multiple functions is shown in Fig. 3.8. In Fig. 3.8, two program versions are shown wherein the program segment on the left
Figure 3.6: An illustration of QDiff scheme, handling simultaneous function insertion and deletion.

shows the older version and the program segment on right shows the newer version. The old program contains Main, A, B, and C functions. In the updated version, two new functions, D and E, are added. Furthermore, new code is inserted in main, shown in grey, and function order is changed from Main, funcA, funcB, and funcC
3.4. QDIFF PATCH CREATION PROCESS

Begin

Step 1
Detect all new, deleted, reordered functions and code fragments using clone detection mechanism.

Step 2
Reorder functions according to old layout and keep slop region in place of deleted functions.

Step 3
Detect all branches from symbol table

Step 4
Mark all new functions and code fragments as non-filled

Step 5
MaxSize[slop region in code] \geq MinSize[Non-filled functions]

Pick the largest slop region and fill it by non-filled functions using (0,1) knapsack algorithm

Update slop region size and mark all functions inserted in slop region as filled.

Change all branches referring to filled functions

Yes
All new functions become filled ?

No

Step 6
Place remaining non-filled functions at the end and update all branches to those functions

Step 7
Insert new code fragment in function if there exists a slop region after function and mark it as filled.

Step 8
Place remaining non-filled code fragments at the end

End

Figure 3.7: Flowchart for fixing branches
3.4. QDIFF PATCH CREATION PROCESS

Figure 3.8: An exemplary scenario showing addition of multiple functions.

to Main, funcB, funcD, funcE, funcC, and funcA. Due to the re-ordering all branching instructions, referring to the reordered functions, are also changed. The QDiff scheme, following the Pseudocode 3.7, reorders the functions according to the old order and pad the new functions at the end. The modification also insert a new call instruction, “call 0xaaf2, in the Main function and a corresponding return instruction in the padded code block (0xaaf2). As a final step, the effected call and jump instructions are updated to reflect the new address. These steps, with final ELF file layout, are illustrated in Fig. 3.9.
3.4. QDIFF PATCH CREATION PROCESS

In this section, we discuss the impact of code change on global variables. In an ELF file, global variables are kept in two separate sections, the data section for the initialized variables and the bss section for the uninitialized variables. Both data and bss sections are kept in heap, which is initialized in the RAM by the initializer function. In a traditional ELF file, data and bss sections are kept back to back, i.e.
in-sequence. Due to this placement, the addition of new variables in the data section will shift the subsequent variables in the data variables and all variables in the bss section. This shift will change the instructions involving the modified variable, due to the change in the variables memory addresses. This leads to a significantly large patch file. Similar is the case of variable(s) deletion.

To address the global variable problem, the QDiff scheme reverses the memory address order for the bss section while maintaining the original order for the data section. Fig. 3.10 shows the position of both data and bss sections for a traditional ELF and the QDiff modified ELF file. As there is free space between data and bss sections, therefore addition or deletion of global variables, either initialized or
uninitialized, will not cause any shift in the variables addresses and their subsequent instructions. Addition, deletion, or re-arrangements of initialized or uninitialized global variables, similar to branch instructions, may result in code shift.

In case of compiler reordering and renaming, the global variable related issues are solved using clone detection. If any variable is deleted, it is removed from the heap, however the location is maintained as a slop region. If a variable is added, QDiff checks all slop regions, picks up the largest slop region and fill the slop region by the new variable, using (0,1) knapsack algorithm. If there is no such type of region, the new variable is added at the end. An initialized global variable is added at the bottom of data section. Whereas an uninitialized global variable is added at the top of bss section.

An exemplary scenario for global variable is shown in Fig. 3.11. The left portion of Fig. 3.11 shows gcc compiled traditional ELF file. In this diagram, two variables in the data section, \textit{Timer Interval} and \textit{Msg Interval}, and two variables in the bss section, \textit{Msg Counter} and \textit{Temperature}, are shown. Memory address range of the data section, the bss section, and available free space is $0x0$ to $0x4f1$, $0x4f2$ to $0x6f1$, and $0x6f2$ to $0x7f1$, respectively. To facilitate global variables, QDiff modifies the memory address for bss section and available free space. The modification is shown in the right portion of Fig. 3.11, wherein the data section memory location is the same as the traditional ELF file. However, both the bss section and the free space are changed with memory ranges of $0x5f2$ to $0x7f1$ and $0x4f2$ to $0x5f1$, respectively. The bss section is reversed and kept as a stack, i.e. new variables are assigned lower memory addresses. Due to this, for the exemplary scenario, the arrangement of variables are reversed, i.e. the \textit{Msg Counter} and \textit{Temperature} are placed at the end.
3.4. QDIFF PATCH CREATION PROCESS

Figure 3.11: Illustration of traditional ELF and QDiff modified data/bss sections.
Figure 3.12: Illustration of global variable addition, both initialized and uninitialized.
of the bss section.

As another exemplary scenario, suppose two new variables are added, one in the data section and the other in the bss section. As explained above, the standard ELF file will add these two new variables at bottom of their respective sections, i.e. *Sense Interval* is added at the bottom of data section, whereas *Pressure* is added at the bottom of bss section. Now assume, after compilation, that the variable layout is changed, with respect to Fig. 3.11, as shown in the left side of Fig. 3.12. Adding a new variable in the data section shifts all of the global variables within the bss section, which further affects all the referring instructions, i.e. the *sts* and *lds* instructions within the code section. The *sts* instruction stores a register value in an address location specified in the instruction. The *lds* instruction loads value from address specified in instruction to a certain register. The QDiff scheme detects the newly added variable and the reorganized variable, using clone detection. The layout of the existing variable *Timer Interval* and *Msg Interval* are rearranged according to the old program order. QDiff adds the new variable, *Sense Interval* at the bottom of the data section and *Pressure* on the top of the bss section. The QDiff scheme also changes the instructions referring to these newly and reordered variables. The final modification of variable and code section is shown in right side diagram of Fig 3.12.

Special consideration must be given for memory access via indirect registers. The indirect registers are used for accessing arrays and large data structure. In QDiff, the addition of new array in the data section follows the traditional ELF method, i.e. placed at the bottom of the section. In case of the bss section, the base address of the array is changed while maintaining the array order. This reduces the complexity of handling indirect registers with offsets. Fig. 3.13 shows sequence of variables...
including array after addition in the bss section of GCC compiled ELF file. Fig. 3.14 shows a sequence of the same variables in the QDiff modified ELF file. The order of arrays is kept the same in both ELF files except that the base addresses are changed. The overall flowchart to handle global variables is outlined in Fig. 3.15.

As QDiff modifies the organization way of the data and bss section, this requires changing the initializer function(.init) function to load correct data in memory. QDiff changes the layout of bss and data in memory to stack and heap fashion which allows to grow data and bss section freely. This reduces shift of all global variables in bss section while a single variable is added in data section. The new initializer function
3.4. QDIFF PATCH CREATION PROCESS

Figure 3.14: Sequence of variables including array in bss section of QDiff modified ELF file.

just reads from bss section in reverse direction. However this does not effect the programming style. The programmer will code the same as he used to code before. QDiff changes the code seamlessly and programmer need not to be aware about ELF file structure.

3.4.3 Indirect Addressing

Indirect addressing is used to access a memory location using registers. Effective address of the memory location is specified in indirect register. For example, Timer.Interval resides in address location 0x00f2 of data section in Fig. 3.16. In code section, this
3.4. QDIFF PATCH CREATION PROCESS

Begin

Step 1: Detect all new, deleted, reordered global (initialized and uninitialized) variables using clone detection.

Step 2: Reorder variables according to old layout and keep slop region in place of deleted variables.

Step 3: Detect all instructions referring to global variables.

Step 4: Mark all new initialized and uninitialized global variables as non-filled.

Step 5: Repeat step 5 for uninitialized global variables.

No

MaxSize[slap region in data] >= MinSize[Non-filled initialized variables]

Yes

Pick the largest slap region and fill it by non-filled initialized variables using (0,1) knapsack algorithm.

Update slap region size and mark all variables inserted in slap region as filled.

Change all instructions referring to filled variables.

Step 6: Repeat step 5 for uninitialized global variables.

Yes

All initialized variables become filled?

No

Step 7: Add remaining non-filled initialized global variables at the end of data section and change all instructions referring to them.

Step 8: Add remaining non-filled uninitialized global variables on top of bss section and change all instructions referring to them.

End

Figure 3.15: Flowchart for fixing global variables.
address location is loaded in indirect register \( ir1 \). As \( ir1 \) contains memory location of \( Timer\_Interval \), all load/store instructions using \( ir1 \) effects \( Timer\_interval \). In sub-sequent instruction, \( st\ r2,\ ir1 \), value of \( Timer\_Interval \) is set by register \( r2 \). Typically, indirect addressing is used for faster access of data. Due to the re-arrangement of bss section, QDiff modifies indirect instructions referring to bss section. To explain the working mechanism of QDiff to handle indirect addressing, we use an exemplary scenario (Fig. 3.16). The left portion of Fig. 3.16 shows the modified global variables, \( Timer\_Interval \) and \( Msg\_Interval \). These two variables are set, in the code section, using indirect registers \( ir1 \) and \( ir2 \), respectively. Using the global variable procedure (explained in the flowchart of Fig. 3.15 and shown in right portion of Fig. 3.16), QDiff changes the memory location of \( Msg\_Interval \). This change which is required because of QDiff’s stack-based bss section invalidates the instructions referring to it. To fix this situation, QDiff detects and modifies such instructions. This modification can be done through two type of instructions. First, by updating indirect registers through an immediate value. Second, updating indirect through another register. QDiff generates control flow graph to detect these two type instructions and modifies them to refer to new location of corresponding variable. The updated instruction is shown in the QDiff modified code section wherein the original “\( mov\ ir2,\ 0x04f2 \)” is changed to “\( mov\ ir2,\ 0x07f0 \)”, \( 0x07f0 \) is the new memory location of \( Msg\_Interval \) variable.

Let us consider another example scenario involving arrays and indirect addressing. This scenario is shown in Fig. 3.17, wherein two arrays \( A[0:1] \) and \( B[0:1] \) are added in the data and bss sections, respectively. As in previous example, the QDiff scheme changes the memory location of the \( B[0:1] \) array, while maintaining the original
Figure 3.16: Illustration of indirect address access and global variables, for traditional and modified ELF files.
Figure 3.17: Illustration of indirect address access and global arrays, for traditional and modified ELF files.
memory order for its element in the memory space. In the code section, these arrays
are accessed through offset addition to the values of the indirect registers. In this
case, QDiff modifies the instructions, that involve the base address. For instance, the
instruction “mov ir2, 0x06f2” is changed to “mov ir2, 0x05ee”, 0x5ee is the base
address of the array B[0 : 1]. However, no change is required for instructions that are
accessing higher index value of array, since QDiff maintains the element order for each
array. For instance, the instructions “st r2, ir” and “st r3, ir + 2” accessing B[0]
and B[1], respectively, remain unchanged. The QDiff algorithm to handle indirect
addressing is shown in flowchart of Fig. 3.18.

In the next iteration of code, two arrays, A[0:1] and B[0:1] are added, A[0:1] is
added in data section and array B[0:1] added in bss section. New arrays are shown in
dark grey color in Fig 3.17. Although the array B[0:1] is added at the end, its order
is maintained same. In the updated code section of Fig. 3.17 shows few instructions
referring the global variables and arrays in data and bss section. Arrays are accessed
through adding offset to the indirect registers. As array order is maintained the
same as in the gcc compiled file, there is no need to change the instructions that
are accessing higher index value of array. In this case, QDiff modifies instructions
dealing with the base address. Other instructions are kept intact. In the code section
of Fig. 3.16 and Fig. 3.17, the ”st” instruction sets different global variable through
different indirect registers, ir1,ir2,ir4...irn. Left side diagram of Fig. 3.16 shows new
gcc compiled file. QDiff detects new and reorganized variables using clone detection
mechanism, then reorganizes the existing variables. Then QDiff adds A[0:1] at the
end of data section and B[0:1] on the top of bss section. After this modification,
QDiff changes indirect instructions referring variables in data and bss section. The
Section 3.4: QDIFF Patch Creation Process

**Step 1**: Generate a control flow graph of the whole program.

**Step 2**: Traverse the control flow graph and keep track of indirect registers.

**Step 3**: If any indirect instruction encountered, find the address stored in the related indirect register.

**Step 4**:
- **Yes**: Calculate the corresponding address in the QDiff modified ELF file for the address stored in the indirect register.
- **No**: Update instructions so that the indirect register contains the address in the QDiff modified ELF file.

**Final Step**: Graph traversing ends?
- **Yes**: End.
- **No**: Continue with the next step.

Figure 3.18: Flowchart for fixing indirect addresses.

The final modification of the variable and code section is shown in the right-hand side diagram of Fig. 3.17.
3.4.4 Relative Jumps

A Relative jump is a position independent code. These jumps are generally smaller in size and are measured relative to current instruction pointer. Jump instructions involve either positive (address is after current instruction address) or negative (address is before current instruction address) offset. Relative jump instructions are changed if the instructions are added or deleted in between the jumping source and destination. The QDiff scheme handles relative jump by the use of basic block calculation technique. The basic block is code that contains at most one branching instruction, generally at the end of code block.

Fig. 3.19 illustrates the changes in the relative jump instructions, due to insertion of new code. Insertion of new block code causes a change in the relative location of the two jump instruction, at address 0x28e and 0x3ae. To address this change, QDiff first identifies all the basic blocks in the program code. Afterward, the basic block which causes least change in the relative jumps is identified and is moved to end of program code. Slop regions are inserted as replacement for the moved block of code. Relative jump requires special care as there might exists some relative jumps in new code that jump to some position in old program. To overcome this issue, QDiff only pads the basic block at the end. The final ELF file, as is modified by the QDiff scheme is shown in Fig. 3.20. QDiff moves the new code block at 0xaaaf2, therefore, maintaining the same address for relative jumps. Flowchart 3.21 describes this procedure to solve relative jump related issues.
3.5 QDiff Patch Deployment Process

The base-station encodes delta and transmits it over the network. Based on requirement, the base-station disseminates delta over the whole network or part of the network. Any transmission protocol format such as deluge, trickle, stream can be used. We assume that the transmission protocol will deliver the patch completely and reliably over the network. The delta produced by QDiff is not dependent on any particular dissemination protocol. Sensor mote receives delta and stores it in internal
3.5. QDIFF PATCH DEPLOYMENT PROCESS

flash or RAM if its size is small. However, if delta size is large, it is stored in external flash memory. When sensor mote receives delta completely, it calls bootloader. Bootloader parses and applies delta on the existing code. If parsing fails, bootloader restores the old image.

Figure 3.20: Illustration of QDiff changes to handle relative jumps.
3.5. QDIFF PATCH DEPLOYMENT PROCESS

Begin

Step 1
Pick a similar function touple computed using clone detection tool

Step 2
Identify all basic blocks

Step 3
Identify shifted relative jumps

Step 4
Calculate delta between old and new function ignoring shifted jump

Step 5
Detect basic blocks in calculated delta

Step 6
Remove and modify other instructions in delta except basic blocks

Step 7
Fix shifted relative jumps

Step 8
Calculate delta between old and new function considering fixed relative jumps

Step 9
Modify code fragment that is padded at the end of program by branch handling algorithm

All similar function touple traversed?

Yes

End

No

Figure 3.21: Flowchart for fixing relative jumps.
3.5. QDIFF PATCH DEPLOYMENT PROCESS

Begin
Fix all branches
Fix all global variables
Fix all relative jumps
Calculate delta for code, data and bss section
Aggregate and optimize all delta
Transmit delta over wireless medium
End

Figure 3.22: Flowchart for QDiff.
Chapter 4

Implementation and Evaluation

In this chapter, we present the implementation details of QDiff, performance metrics, experimental use cases and their results. For performance comparison, we considered Zephyr [51] and Hermes [49], as they are the latest work on differential reprogramming and show better efficiency than all previous schemes. Furthermore, both qualitative comparison and overheads are also investigated.

4.1 Implementation details

The proposed QDiff scheme for over-the-air reprogramming of motes is implemented using the Java language, to run on the WSNs base-station. The IRIS platform, based on Atmega-128 microcontroller with 128KB internal flash and 4KB RAM, is used as motes for WSNs. The motes use TinyOS as its operating system wherein the applications are written using nesC programming language [21]. In order to create a single OS image, i.e. monolithic image, the nesC compiler converts the nesC code to a corresponding C code. This C file is later compiled and linked by an architecture specific C compiler, e.g. avr-gcc for the IRIS mote. We make use
of the Bauhaus toolkit [55] to detect clones between programs. Bauhaus is a well-known clone detection tool and works for various programming languages, including C. While the Bauhaus toolkit does not work directly for assembly language, it is still used indirectly as the compiler keeps the same identifier names for both C and Assembly languages. We have extended the Bauhaus toolkit to enhance the clone detections, especially to support “Type 3” clone detection. To determine clones, various ELF sections, i.e. code, bss and data are dumped using standard binary utilities tools, e.g. avr-objdump and avr-readelf for the IRIS mote. The implemented QDiff code using the three sections, i.e. code, bss and data obtained for the old and new ELF program, determines the patch which is then transmitted to the motes, over the air. Google-guava [2] API and extended Google diff-match-patch [1] are used for analyzing dumped ELF files and determining the patch.

The generated patch is made of series of commands as follows:

```
insert <newOffset><size=s1><data>
delete <newOffset> <size=s2>
copy <oldOffset><size=s3><newOffset>
pad <newOffset><size=s4>
repeat <newOffset><size=s5><numrepeat=n><addr1> ... <addrn>
```

**insert**: This command inserts data of size s1 at newOffset memory location. This data can either be the machine code or global variables.

**delete**: This command erases code and variable of size s2 from newOffset memory location.

**copy**: This command copies code and variable from the old ELF to the revised version. The address for the old and the new ELF location is specified by oldOffset
and \textit{newOffset}, respectively. The data size to be copied is specified by the size \(s3\).

This command is also used by the Hermes and Zephyr schemes to build the image for the external memory.

\textbf{pad}: This command pads \(s4\) many zero, in new image, at memory location specified by \textit{newOffset}.

\textbf{repeat}: This command performs consecutive copy of an instruction with same size. The number of times and amount of data to be copied is specified by \(n\) and size \(s5\), respectively. This command is used by the zephyr or hermes schemes for transmitting jump table.

The command opcode and offset address are represented using one and two bytes, respectively. We assume address length of 24 bits which can address up to 16MB and 32MB if the address is byte aligned and word aligned, respectively. The generated patch is fragmented into network packets and disseminated over the network using the dissemination protocol, e.g. Stream. The received patch, depending on its size, is stored either in the RAM (if smaller than available RAM space) or the external memory. The external memory is used only when the patch size exceeds the RAM space. We have implemented the bootloader using nesC language. The bootloader parses the received patch, builds target flash memory pages and writes them back to the internal flash memory.

### 4.2 Evaluation Scenarios

For experimental purposes, we use IRIS sensor mote. It uses atmega1281 microcontroller as a processing unit. IRIS has 8KB RAM, 128KB internal flash and 512KB external flash memory. For debugging purpose, it provides three LEDs, LED0, LED1,
LED2. Fig. 4.1 shows an IRIS mote. To cover various code modifications and to evaluate the performance of reprogramming protocols, we ran numerous evaluation cases. These cases are explained in the later part of this chapter. A comprehensive evaluation of a differential scheme needs to cover all the four major issues (branches, global variables, indirect addressing, relative jumps) that are described in Chapter 3. To this end, we introduce the following 8 cases. Case 1 only requires simple difference calculation, however, all other cases (Case 2 - Case 8) cover these four major issues. While the chosen code is small, in large code the same four issues will simply be repeated multiple times.

Case 1: Blink to Blink with timer value changed
4.2. EVALUATION SCENARIOS

Blink is an well known application that toggles LEDs of sensor mote after a certain time interval. This time interval is maintained through a timer module. The Blink application considered here toggles the first LED, i.e. LED1 of mote. In the first version of Blink, the timer value is set to 250 ms and in later version of Blink, timer value is set to 500 ms. Fig. 4.2 illustrates code changes in case 1.

Case 2: Blink to Blink with two LED toggles in timer1
In this case, we consider Blink with three timers: Timer0, Timer1 and Timer2. Each timer corresponds to a LED in iris mote: LED0, LED1 and LED2. Timer intervals are set to 250 ms, 500 ms and 1000 ms for Timer0, Timer1 and Timer2, respectively. When a particular timer fires, the corresponding LED is toggled. In modified Blink, a single line is added in Timer0 function. Modified Timer0 function toggles both LED0 and LED1 simultaneously. Fig. 4.3 illustrates code changes in case 2.

Case 3: Blink to Blink with two toggles in each timer modified Blink
In this case, two LEDs are toggled simultaneously in each timer function. In modified timer functions, Timer0 toggles LED0 and LED1, Timer1 toggles LED1 and LED2, Timer2 toggles LED2 and LED0. Fig. 4.4 illustrates code changes in case 3.

Case 4: Blink with two toggles in each timer to further modification in Blink
This is further update of modified blink in case 3. In this case each timer toggles LED(s) but these toggles are reorganized i.e. Timer0 toggles LED2 and LED0, Timer1 toggles LED2 and LED1, Timer2 toggles LED1 and LED0. Fig. 4.5 illustrates code changes in case 4.

Case 5: RadioCountToLeds to RadioCountToLeds showing least significant two bits
RadioCountToLeds is a widely used application for checking Active Message (AM) [62] communication. Base-station broadcasts a counter value as AM packet, sensor mote receives this packet and displays last three bit on its LEDs. We modify RadioCountToLeds as it displays last two bits on its LEDs and value displayed in third LED is discarded. Fig. 4.6 illustrates code changes in case 5.

**Case 6: RadioCountToLeds to RadioCountToLeds showing complement of least significant three bits**

In this case, RadioCountToLeds is modified to displays 1’s complement of least significant three bits of received counter on sensor mote LEDs. Fig. 4.7 illustrates code changes in case 6.

**Case 7: RadioCountToLeds to RadioCountToLeds showing most significant three bits**

RadioCountLeds displays least significant three bits of received 16 bit counter value on sensor mote LEDs. In this case, RadioCountToLeds is modified to display most significant three bits of received counter value. Fig. 4.8 illustrates code changes in case 7.

**Case 8: RadioCountToLeds to RadionCountToLeds showing even numbers.**

In this case, RadioCountToLeds is modified to display counter bits, if it has an even value, otherwise no change is made in sensor mote LEDs. Fig. 4.9 illustrates code changes in case 8.
Figure 4.2: Case 1: Timer value is changed.
Figure 4.3: Case 2: Addition of one toggle in Timer0.
event void Timer0.fired()
    {
      call Leds.led0Toggle();
    }

event void Timer1.fired()
    {
      call Leds.led1Toggle();
    }

event void Timer2.fired()
    {
      call Leds.led2Toggle();
    }

Figure 4.4: Case 3: Addition of one toggle in each timer i.e. Timer0 toggles LED0 and LED1, Timer1 toggles LED1 and LED2, Timer2 toggles LED2 and LED0.
event void Timer0.fired()
{
    call Leds.led0Toggle();
    call Leds.led1Toggle();
}

event void Timer1.fired()
{
    call Leds.led1Toggle();
    call Leds.led2Toggle();
}

event void Timer2.fired()
{
    call Leds.led2Toggle();
    call Leds.led0Toggle();
}

Blink with two toggles

Figure 4.5: Case 4: Toggles are reorganized i.e. Timer0 toggles LED2 and LED0, Timer1 toggles LED2 and LED1 , Timer2 toggles LED1 and LED0.
4.2. EVALUATION SCENARIOS

RadioCountToLeds

radio_count_msg_t* rcm = (radio_count_msg_t*)payload;
if (rcm->counter & 0x1) {
call Leds.led0On();
} else {
call Leds.led0Off();
}
if (rcm->counter & 0x2) {
call Leds.led1On();
} else {
call Leds.led1Off();
}

RadioCountToLeds showing two LSB

radio_count_msg_t* rcm = (radio_count_msg_t*)payload;
if (rcm->counter & 0x1) {
call Leds.led0On();
} else {
call Leds.led0Off();
}
if (rcm->counter & 0x2) {
call Leds.led1On();
} else {
call Leds.led1Off();
}
if (rcm->counter & 0x4) {
call Leds.led2On();
} else {
call Leds.led2Off();
}

Figure 4.6: Case 5: RadioCountToLeds is modified to display last two bits on its LEDs and value displayed in third LED is discarded.
4.2. EVALUATION SCENARIOS

RadioCountToLeds

\[
\text{radio\_count\_msg\_t* rcm = (radio\_count\_msg\_t*)payload;}
\]

\[
\text{if (rcm->counter & 0x1) {}
\text{call Leds.led0On();}
\text{}}
\]

\[
\text{else {}
\text{call Leds.led0Off();}
\text{}}
\]

\[
\text{if (rcm->counter & 0x2) {}
\text{call Leds.led1On();}
\text{}}
\]

\[
\text{else {}
\text{call Leds.led1Off();}
\text{}}
\]

\[
\text{if (rcm->counter & 0x4) {}
\text{call Leds.led2On();}
\text{}}
\]

\[
\text{else {}
\text{call Leds.led2Off();}
\text{}}
\]

RadioCountToLeds showing three LSB

\[
\text{radio\_count\_msg\_t* rcm = (radio\_count\_msg\_t*)payload;}
\text{rcm->counter=\neg(rcm->counter);}
\]

\[
\text{if (rcm->counter & 0x1) {}
\text{call Leds.led0On();}
\text{}}
\]

\[
\text{else {}
\text{call Leds.led0Off();}
\text{}}
\]

\[
\text{if (rcm->counter & 0x2) {}
\text{call Leds.led1On();}
\text{}}
\]

\[
\text{else {}
\text{call Leds.led1Off();}
\text{}}
\]

\[
\text{if (rcm->counter & 0x4) {}
\text{call Leds.led2On();}
\text{}}
\]

\[
\text{else {}
\text{call Leds.led2Off();}
\text{}}
\]

Figure 4.7: Case 6: RadioCountToLeds is modified to displays 1’s complement of least significant three bits of received counter.
4.2. EVALUATION SCENARIOS

Figure 4.8: Case 7: RadioCountToLeds is modified to display most significant three bits of received counter value.
RadioCountToLeds

Figure 4.9: Case 8: RadioCountToLeds is modified to displays counters bits if it has an even value, otherwise no change is made in sensor mote LEDs.
4.3 Quantitative Performance Metrics

Energy is the most important resource in wireless sensor networks. Time required for reprogramming sensor networks is another important factor as sensor networks are not able to do their intended work during reprogramming period. The following three metrics mainly affect energy and time in WSN reprogramming.

1. **Patch size**: Patch size refers to the file size of a generated patch. The generated patch includes both data and commands, as explained in Section 4.1. The size however, does not include the overhead that may be caused as the patch size is fragmented into smaller network packets. Patch size is an important performance metric as smaller patches imply lesser data transmission in WSNs, hence lower energy consumptions. Furthermore, small patch size also implies lesser reprogramming time for each mote.

2. **Internal flash writing**: Internal flash writing refers to the number of flash memory pages required to be rewritten upon receiving a patch. The internal flash size also includes the overhead of using external flash to build (or modify) the newly received image. However, we do not include the overhead to load the reprogramming protocol from the external storage into internal flash. Energy consumption of flash memory writing is similar to wireless transmissions. Therefore, reducing internal flash writing reduces the energy consumption of mote, hence allowing more of the motes reprogramming while maintaining the network lifetime.

3. **External flash usage**: External flash usage refers to the amount of external flash memory required for storing patch script and building new image from
patch script. External flash, if present, for certain applications is mainly used for various network functions, e.g. storing sensing and debugging information, logging network events etc. However, it is not a mandatory requirement for other applications, e.g. real-time pressure sensing in home automation. Moreover, accessing the external flash memory for either read or write, performed via SPI bus, is both slow and power hungry. Therefore, reducing the amount of external flash, or ideally eliminate the need of it by the reprogramming scheme is of great importance for the WSNs.

4.4 Performance Evaluation

In this section, we quantitatively and qualitatively analyze Stream, Hermes, and QDiff reprogramming schemes for various evaluation scenarios (Section 4.2) using various performance metrics (Section 4.3).

4.4.1 Patch size

Patch size is the file size of a generated patch. The lower the size, the better the scheme is. Patch sizes for all of the evaluation scenarios, for Stream, Hermes and QDiff, are shown in Fig. 4.10. QDiff outperforms Stream and Hermes up to 250 and 12 times, respectively, for all of the scenarios considered. The highest relative improvement, between QDiff and Hermes, is observed for “Case 8”. In this case, due to compiler optimizations, the function order has been changed. This is not detected by the Hermes scheme. QDiff, on the other hand, using the clone detection technique is able to detect this structural change, therefore is much more efficient.
4.4. PERFORMANCE EVALUATION

Figure 4.10: Patch size comparison for the Stream, Hermes and QDiff reprogramming schemes.

4.4.2 Internal flash writing

Internal flash writing refers to the number of flash memory pages required to be rewritten upon patch receiving. Flash memory writing is page based. In IRIS mote, size of a flash page is 256 bytes [65]. Experimental results for internal flash writing for all evaluation scenarios are shown in Fig. 4.11. Byte equivalents of internal flash writing are shown in Table 4.1. Both Stream and Hermes schemes require complete flash rewriting of the program, and hence require the same amount of flash page writing. Stream also requires to rewrite the reprogramming protocol in internal flash, and Hermes requires to rebuild the code image on external flash memory. In more
complex real-life scenarios, these flash writings could be significantly higher. As shown in Table 4.1, the proposed QDiff scheme outperforms both Stream and Hermes, by a factor of 21 times. Furthermore, the QDiff scheme, unlike other schemes, is not affected by the complexity of the use case. For instance, the frequency of internal flash writings for both Stream and Hermes jump five times, between Case 4 and Case 5. This is because, QDiff only writes modified flash pages, whereas both Stream and Hermes rewrite the whole program flash every time.
Table 4.1: Internal flash memory bytes comparison for the Stream, Hermes and QDiff

<table>
<thead>
<tr>
<th>Case</th>
<th>Stream</th>
<th>Hermes</th>
<th>QDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2304</td>
<td>2304</td>
<td>256</td>
</tr>
<tr>
<td>2</td>
<td>2304</td>
<td>2304</td>
<td>512</td>
</tr>
<tr>
<td>3</td>
<td>2560</td>
<td>2560</td>
<td>512</td>
</tr>
<tr>
<td>4</td>
<td>2560</td>
<td>2560</td>
<td>512</td>
</tr>
<tr>
<td>5</td>
<td>11008</td>
<td>11008</td>
<td>512</td>
</tr>
<tr>
<td>6</td>
<td>11008</td>
<td>11008</td>
<td>512</td>
</tr>
<tr>
<td>7</td>
<td>11008</td>
<td>11008</td>
<td>512</td>
</tr>
<tr>
<td>8</td>
<td>10752</td>
<td>10752</td>
<td>512</td>
</tr>
</tbody>
</table>

4.4.3 External flash usage

External flash usage refers to the amount of external flash memory required for storing patch script and building new image from that script. In the Hermes scheme, the bootloader builds the new image using the patch script on external flash. Similarly, the Stream scheme also stores both the reprogramming protocol and the patch file in the external flash. Experimental results for external flash memory usages(in pages) and their byte equivalents are shown in Fig. 4.12 and Table 4.2, respectively. Results do not include the flash memory required for storing the golden image. Both Stream and Hermes require large volume of external memory. Hermes requires more external flash as it needs to store the patch script, as well as build and store new image. However, the QDiff scheme does not require any external memory as it can build the new image using RAM. This results in significant lower energy consumption and less reprogramming time for the motes.

4.4.4 System Restart

Existing reprogramming schemes require the mote to be restarted. This restart is essential as these schemes perform complete code flash rewriting, even for the simplest
4.4. PERFORMANCE EVALUATION

Figure 4.12: External flash memory pages comparison for the Stream, Hermes and QDiff reprogramming schemes.

<table>
<thead>
<tr>
<th>Case</th>
<th>Stream</th>
<th>Hermes</th>
<th>QDiff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>2120</td>
<td>2157</td>
<td>0</td>
</tr>
<tr>
<td>Case 2</td>
<td>2292</td>
<td>2563</td>
<td>0</td>
</tr>
<tr>
<td>Case 3</td>
<td>2316</td>
<td>2687</td>
<td>0</td>
</tr>
<tr>
<td>Case 4</td>
<td>2312</td>
<td>2546</td>
<td>0</td>
</tr>
<tr>
<td>Case 5</td>
<td>10766</td>
<td>11515</td>
<td>0</td>
</tr>
<tr>
<td>Case 6</td>
<td>10810</td>
<td>11592</td>
<td>0</td>
</tr>
<tr>
<td>Case 7</td>
<td>10782</td>
<td>11575</td>
<td>0</td>
</tr>
<tr>
<td>Case 8</td>
<td>10690</td>
<td>12148</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: External flash memory bytes comparison for the Stream, Hermes and QDiff.
4.4. PERFORMANCE EVALUATION

4.4. PERFORMANCE EVALUATION

Restarting of the WSN mote is energy inefficient and results in overall network instability and functional failure. For instance, in a recent deployment on Reventador Volcano, reboots led to 3-day network outage, reducing mean node uptime from >90% to 69% [14, 64]. An exception is the Elon scheme. However, Elon only works for certain WSNs platform, e.g. TelosB. Our proposed QDiff scheme does not require mote restart and is independent of the WSNs platform and its operating system.

4.4.5 Code Overheads

**Code size:** A reprogramming scheme usually inserts new instructions, slop regions, jump tables and other translation schemes, which result into large program sizes. Maintaining lower code size increment is important as the mote has a limited available memory. The proposed QDiff reprogramming scheme also introduces overhead as additional jump calls. However, such overhead is insignificant as each of these jump instruction, and subsequent return instruction, take only 6 bytes. Furthermore, as the call-return instructions are only added for only few functions, the overall increment is, on average, within 3% of the total code memory.

**Fragmentation:** After several reprogramming the program code residing on the internal flash memory, due to the added call-return by the QDiff scheme, may get fragmented. To overcome this issue the base-station keeps track and instructs motes to rebuild the new program image if fragmentation reaches a certain threshold. In the new code, the code blocks addressed by the call-return functions are replaced to the callee locations. The defragmentation process is not energy intensive since the base-station only sends the fragmented call-jump code over the network.
### 4.4. PERFORMANCE EVALUATION

#### 4.4.6 Heterogeneity

Many different hardware platforms and operating systems are available for WSNs, with diverse specifications. All of the existing schemes, by design, can only support a subset of these platforms. For instance, some of the reprogramming schemes [14, 48] are specific to the TinyOS operating system, several others [15, 16] are specific to the Contiki operating system whereas some [14] are tied to a particular mote platform. However, the proposed QDiff is independent of the mote platform and its operating system.

An overall qualitative comparison of Deluge, Rsync, Zephyr, Hermes, Elon and QDiff is shown in Table 4.3.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Patch size</th>
<th>External flash</th>
<th>Flash rewrite</th>
<th>Heterogeneity</th>
<th>System Restart</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deluge [26]</td>
<td>Huge</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
<td>Yes</td>
</tr>
<tr>
<td>Rsync [60]</td>
<td>Medium</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
<td>Yes</td>
</tr>
<tr>
<td>Zephyr [51]</td>
<td>Low</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
<td>Yes</td>
</tr>
<tr>
<td>Hermes [49]</td>
<td>Low</td>
<td>Yes</td>
<td>Yes</td>
<td>Multiple</td>
<td>Yes</td>
</tr>
<tr>
<td>Elon [14]</td>
<td>Medium</td>
<td>Yes</td>
<td>Yes</td>
<td>Single</td>
<td>No</td>
</tr>
<tr>
<td>QDiff</td>
<td>Very low</td>
<td>No</td>
<td>No</td>
<td>All</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 4.3: Qualitative comparison of various reprogramming schemes
Chapter 5

Conclusion

Wireless reprogramming is an important requirement for managing large scale wireless sensor networks. Existing reprogramming schemes transmit large amounts of information over the network, rewrite the whole internal flash memory and require large amount of external flash memory. Radio transmission and flash memory writing are two of the most energy- and time-intensive operations in wireless sensor networks. This thesis describes, QDiff, a new reprogramming scheme, which reduces the amount of information to be transmitted, reduces internal flash memory writing and eliminates necessity of external memory without modifying the compiler. Experiments show that QDiff reduces information size by a factor of 11 times and internal flash memory writing by a factor of 20 times than existing reprogramming schemes. QDiff achieves this efficiency by exploiting the strength of clone detection mechanisms and re-arranging global variables in the final executable file. To the best of our knowledge, QDiff is the first scheme utilizing clone detection in high level language and exploits clone information in machine language for efficient reprogramming in sensor networks. Furthermore, QDiff works on various hardware and software platforms to support the application-specific nature of sensor networks.
Our future work on QDiff focuses on fault-tolerance support while using QDiff as a reprogramming scheme. Updating software using faulty patch might result crash of sensor node forever. It is an important issue as sensor-nodes run without human interaction. We are currently looking for a way that is compatible with QDiff, as well as able to provide fault-tolerance support by recovering software from golden image after system crash without rebooting sensor node.
Bibliography


BIBLIOGRAPHY


