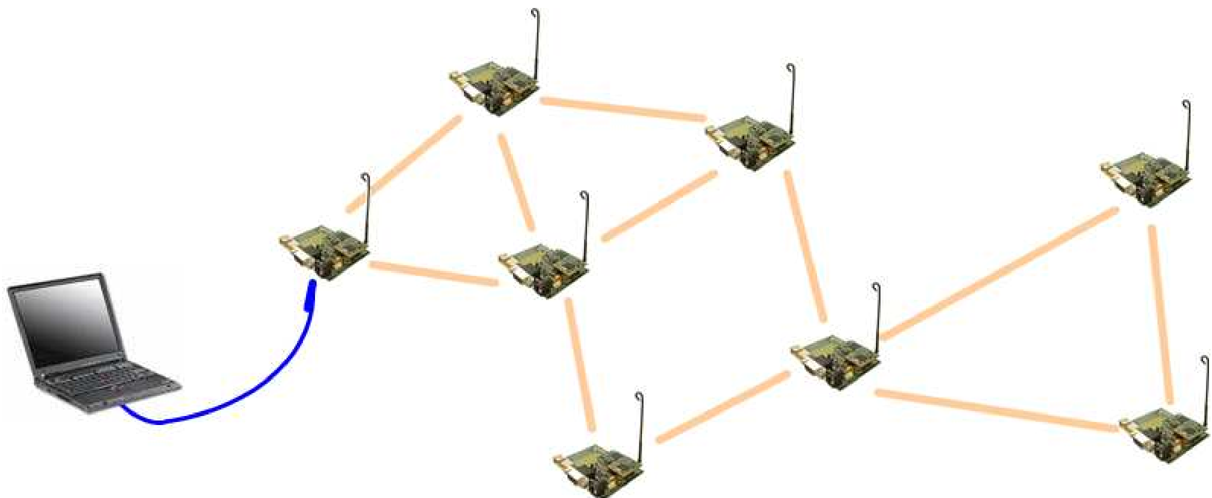
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing Group

Master Thesis

Updating Wireless Sensor Networks



Andreas Pfenninger

{ apfenninger@student.ethz.ch }

Winter, 2006/2007

Prof. Dr. Roger Wattenhofer
Advisor: Pascal von Rickenbach

Distributed Computing Group
Department of Computer Science
Swiss Federal Institute of Technology (ETH) Zurich

Abstract

This thesis describes a mechanism to update large wireless sensor networks. Due to the dynamic nature of such networks, sensor nodes occasionally have to be reprogrammed, especially for design-implement-test iterations. Manually reprogramming every node by physically reaching it is a very cumbersome task, and may be infeasible if nodes of the network are unreachable. Therefore, a wireless update mechanism is needed. Exchanging the running application on a node by transmitting the complete program image is not efficient for small changes in the code. It consumes a lot of bandwidth and time. The goal of this thesis is to use an incremental network programming approach to minimize the transmitted code size. A difference file for the new code is computed using the Xdelta algorithm, and is then distributed over the network. The delta is decoded on the node to build and install the new application. The proposed update protocol has been implemented for the Tinynode sensor nodes, which run TinyOS, a component-based operating system for highly constraint embedded platforms.

Preface

This thesis is submitted for partial fulfillment of the requirements of the degree Master of Science in Computer Science at the Swiss Federal Institute of Technology (ETH) Zurich. The thesis was derived during a six month project from June 6th to December 5th 2006 in the Distributed Computing Group headed by Prof. Dr. Roger Wattenhofer at the Computer Engineering and Networks Laboratory of the Information Technology and Electrical Engineering Department of the Swiss Federal Institute of Technology Zurich.

Acknowledgments

I would like to thank all the people who supported me during the course of this work and made this thesis possible. First I like to thank the people of the Distributed Computing Group for a great time. Special thanks go to my advisor Pascal for his good ideas and feedback and his invaluable support.

Throughout my studies my family has accompanied me through ups and downs and has given me support and care all the time. My friends continuously give me confidence, for which I am very grateful. Michi has always been there for me, her understanding, encouraging and love has given me the strength needed to keep up.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

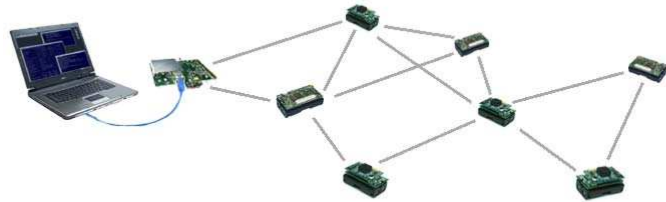
Distributed Computing Group

Prof. Dr. Roger Wattenhofer
http://www.dcg.ethz.ch

Diploma/Master's Thesis "Updating Wireless Sensor Networks"

Recently, increasing research attention has been directed toward wireless sensor networks. Large numbers of small, inexpensive devices that integrate sensing, computation and communication will be monitoring environmental changes, water contamination, seismic activity etc. These sensor nodes exhibit multiple constraints such as limited CPU power, narrow bandwidth, and limited energy budget.

There are many reasons nodes occasionally have to be reprogrammed. An important one is that applications go through a number of design-implement-test iterations during the development cycle. It is clear that it is highly impractical to physically reach all nodes in a network, so wireless updating scheme is required.



In this thesis we will develop a new update mechanism, aimed at the requirements and restrictions specific to wireless sensor networks. In general, such an updating procedure consists of three steps: encoding, dissemination and decoding of a new program code. We will consider all three steps in detail to come up with a tailored solution. In a second step the proposed update protocol will be implemented for the mica2 motes. These quasi-standard sensor nodes run TinyOS, an operating system for highly constraint embedded platforms.

The goal of this thesis is to design and implement a new wireless updating scheme for sensor networks. The solution should take the specific characteristics of such networks into account. Secondly, the obtained solution should be implemented for the TinyOS platform. Finally, we are intressted in comparing this solution with already existing work in this field.

Required

- Advanced Programming skills.
- Basic C knowledge.
- Interest in designing and implementing new protocols on an embedded platform.

Contacts

- Pascal von Rickenbach, pascalv@tik.ee.ethz.ch, ETZ G61.3, phone 27007
- Roger Wattenhofer, wattenhofer@tik.ee.ethz.ch, ETZ G61.4, phone 26312

Contents

List of Figures xi

List of Tables xiii

1 Introduction 1

1.1 Wireless Sensor Networks 1
1.2 TinyOS 2
1.3 TinyNode 2
1.4 Motivation 3

2 Related Work 5

2.1 Network Programming 5
2.1.1 XNP 5
2.1.2 Deluge 6
2.2 Differential Network Programming 8
2.2.1 Reijers and Langendoen 8
2.2.2 Multi-hop Over-The-Air-Programming (MOAP) 9
2.2.3 Jeong and Culler 9
2.2.4 Koshy and Pandey 10
2.2.5 FlexCup 11
2.2.6 Dressler and Truchat 12
2.3 Dissemination in Wireless Sensor Networks 12
2.3.1 Trickle 12
2.3.2 Sprinkler 12
2.3.3 Infuse 12
2.3.4 MNP 13

2.3.5 Gappa 13
2.3.6 Aqueduct 13
2.4 Virtual Machine Approach 14
2.4.1 Maté / Bombilla 14
2.4.2 SensorWare 15

3 Analysis **17**
3.1 Requirements 17
3.2 Wireless Dissemination 18
3.3 Encoding 18
3.3.1 Diff 19
3.3.2 Rsync 19
3.3.3 Xdelta 19
3.3.4 Zdelta 21
3.3.5 VCDIFF 21
3.4 Update Dissemination and Feedback 22
3.5 Code Recombination 22
3.5.1 Halve Memory 22
3.5.2 2 Phase Approach 22
3.5.3 Build in EEPROM 23

4 Design and Implementation **25**
4.1 Wireless Dissemination 25
4.2 Encoder 27
4.3 Update Installation 27
4.4 Decoder 28

5 Results **31**
5.1 Size of Code Updates 31
5.2 Time Used for Updates 32
5.2.1 Star Topology 32
5.2.2 Line Topology 34

6 Conclusions and Future Work **37**
6.1 Summary 37
6.2 Evaluation and Discussion 37
6.3 Future Work 38

A Software Manual **39**
A.1 Installing the Boot Loader 39
A.2 Installing DelugeDiff 40
A.3 Reprogramming with a New Program Image 40
A.4 Updating with a New Program Image 41
A.5 Frequently Asked Questions 43

Bibliography **45**

List of Figures

1.1 A typical sensor node 2
1.2 The TinyNode sensor node 3
1.3 TinyNode module placement 4

2.1 Network programming 6
2.2 Spatial multiplexing in Deluge 7
2.3 Address shifts due to inserted code 8
2.4 Number of instructions affected by address shifts 9
2.5 Memory layout for diff-based approach 10
2.6 Memory layout for slop space approach 11
2.7 Forwarding bridge in Aqueduct 14

3.1 Pseudo-code for the Xdelta algorithm 20

4.1 Memory layout of Deluge image 26
4.2 Deluge Image Descriptor struct 26
4.3 The decoding process 28
4.4 EEPROM Storage component 29

5.1 Test setup with star topology 33
5.2 Test setup with line topology 34

List of Tables

5.1 Code size of delta patches for different update scenarios 32
5.2 Update time for Blink to BlinkFast in star scenario 33
5.3 Update time for Blink to CntToLeds in star scenario 33
5.4 Update time for Blink to Oscilloscope in star scenario 34
5.5 Update time for Blink to BlinkFast in line scenario 34
5.6 Update time for Blink to CntToLeds in line scenario 35

1

Introduction

1.1 Wireless Sensor Networks

In our everyday life, we encounter sensors of all different kinds without even taking notice of. Motion sensors turn on lights when we walk by, the heating or air conditioning of rooms is controlled by temperature sensors and fire detectors alert us in case of emergency.

Recently, a lot of attention has been directed toward extended, "intelligent" sensors, that can not only conduct certain measurements, but are equipped with computational power and over-the-air communication. A lot of additional application areas have appeared for these new devices, ranging from medical applications, home automation, traffic control and monitoring of eco-systems to security and surveillance applications.

The development of these enhanced sensors is a logical consequence of the continuing technological progress. Moore's Law states that the transistor density of integrated circuits doubles about every 24 months. The law still fulfills it's prediction year after year, leading to more powerful and smaller devices. The low power and miniature embedded processors, radios, sensors and actuators are often integrated on a single chip, and are relatively cheap. The combination of sensor, micro controller and radio transceiver is often referred to as **sensor node**. A picture of a typical sensor node is shown in Figure 1.1. To take full advantage of theses new devices, they are often used in large numbers, resulting in systems that are called wireless sensor networks. These networks differ considerably from current networked and embedded systems. They combine the large scale and distributed nature of networked systems such as the Internet with the extreme energy constraints and physically coupled nature of embedded control systems. Since the CPU power, battery lifetime, memory size and radio bandwidth are inherently constrained resources, the design of wireless sensor networks requires a proper understanding of the interplay between network protocols, energy-aware design, signal-processing algorithms and distributed programming.

The concept of miniaturization has been pushed to the limit with the vision of "Smart Dust"

Figure 1.1: A typical sensor node (cite).

[WLLP01], that could enable real "Smart Environments". The sensor nodes should be scaled down to the size of a dust particle, and manufactured in large numbers, to be distributed over an area like dust, and remain functionally as long as possible.

1.2 TinyOS

TinyOS [HSW^+00] is an open-source embedded operating system designed for wireless sensor networks. It features a component-based architecture that takes into account the inherent memory constraints of sensor networks. TinyOS is programmed in **NesC**, a language derived from C and specially tailored toward sensor nodes. It applies an event-driven execution model, that flexibly integrates into the unpredictable nature of wireless communication and physical world interfaces by enabling fine-grained power management. There is no traditional kernel, process management or memory management. Dynamic data structures are not supported, since there is no possibility to dynamically allocate memory. Multithreading is only possible through events. Basic functionality as well as certain applications for the sensor nodes are contained in the component library, including timer interfaces, sensor drivers, network protocols, distributed services and data acquisition and storage tools. Most of the tools are completed with Java utilities for cooperation with desktop computers. At compile time, the TinyOS code is statically linked with program code, and compiled into a small binary, using a custom GNU tool chain.

The large open-source community around TinyOS has ported the system to numerous platforms and sensor boards, the most popular being the Mica mote family and Telos nodes. Most current research projects on wireless sensor networks rely on TinyOS and keep on contributing, making it the standard operation system for sensor nodes.

1.3 TinyNode

The sensor node platform used in this thesis is the TinyNode [DFFMM06]. It has been developed and is manufactured by Shockfish SA. The platform is based on a MSP430 micro controller and a Xemics XE1205 radio transceiver. The design philosophy behind the node

architecture was to integrate the core components on a small printed circuit board, and place additional functionality on extension boards. The platform fully supports TinyOS and implements an own network stack. A picture of the TinyNode can be seen in Figure 1.2.

Figure 1.2: A TinyNode sensor node [SA05].

The 8 MHz MSP430 micro controller from Texas Instruments comes with 10K bytes of RAM, 512K bytes of external flash memory and 48K bytes of program space. 19 configurable I/O pins offer up to 6 analog inputs, up to 2 analog outputs and a serial interface. The RS-232 interface and Jtag, that is principally used for debugging, are present on the extension board. The Xemics XE1205 wireless transceiver offers an output power of 0 to +12 dBm and a data rate of 1.2 - 152.3 kbps. It operates on 2 - 10 channels at 868 - 870 MHz. At 76.8 kbps, it reaches a range of 40 m indoors and up to 200 m outdoors. The node is based on an ultra low power 3 V design. The placement of the modules on the board is shown in Figure 1.3.

1.4 Motivation

There are many reasons sensor nodes occasionally have to be reprogrammed, for example for updates of the running program. An additional module may have to be added to the program, or a complete protocol implementation exchanged. Another important reason is that applications go through a number of design-implement-test iterations during the development cycle. It is highly impractical to physically reach all nodes in a network and manually reprogram them by attaching the node to a laptop or PDA, especially for a large number of distributed sensors. It may also be simply infeasible in various scenarios, if the nodes are located in areas that are inaccessible to deployers.

Therefore, a wireless updating scheme is required to set all the nodes up to date with the new version of the application. Another consideration is the amount of code transferred. While it is normal to send the whole code if the application needs to be replaced, it does not make much sense in other cases. If we just add or exchange a part of the code, we transmit code that is already available on the node, maybe just shifted from its original location in program memory by a certain offset. Also if a bug has been identified and fixed in the test process, the biggest part

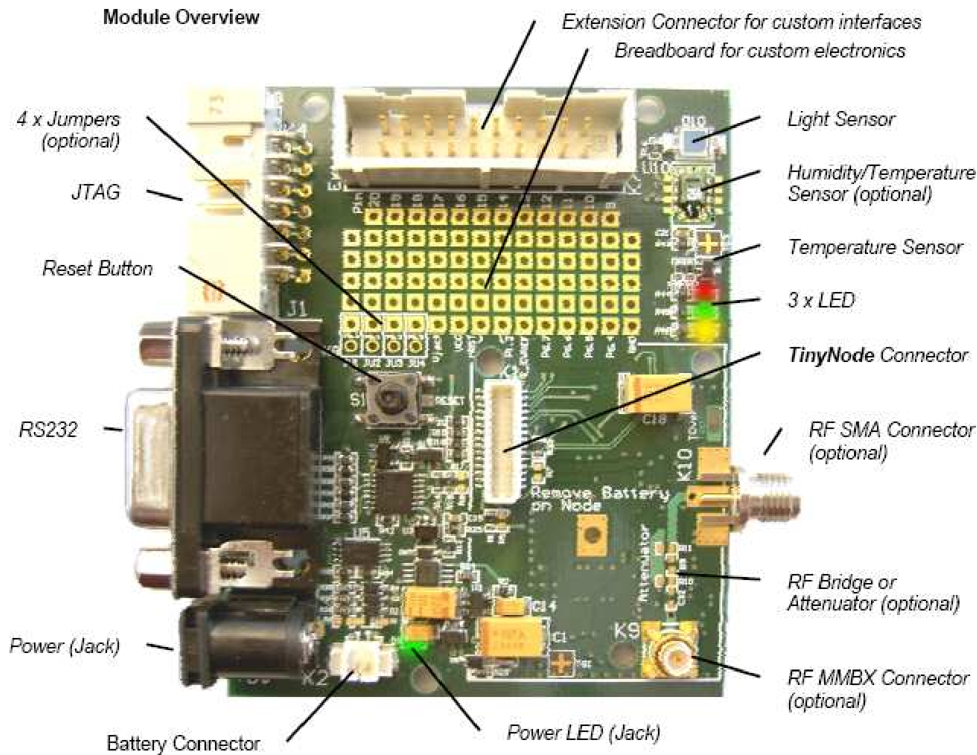


Figure 1.3: The placement of the TinyNode modules on the node [SA05].

of the code remains exactly the same, probably only differing for some functions or constants. To reduce this redundancy, it is much more efficient in terms of used bandwidth and time to only send the changes in the code, and leave the recombination of the new code to the node itself.

This thesis will deal with the design and implementation of such a system for wireless sensor networks. In Section 2, we will give an overview of existing approaches to the problem, ranging from single-hop reprogramming over multi-hop reprogramming to complete virtual machine approaches. Section 3 gives a detailed analysis of the requirements and discusses various encoding algorithms. The implementation of the system is outlined in Section 4, explaining the architecture and design decisions taken. A discussion of the developed system can be found in Section 5, together with test results. Finally, we draw a conclusion and give indications on future work that could be done in Section 6. A software user manual with detailed instructions on how to use the program is contained in the appendix A.

2

Related Work

In this section, we will summarize some previous work on updating wireless sensor networks that has been presented over the last years. We will start with the first straight forward solutions, that completely exchange the affected program images, in Section 2.1. Afterwards, we will take a closer look at the more sophisticated differential approaches in Section 2.2, and also outline some theoretical papers written on dissemination of data in wireless sensor networks in Section 2.3.

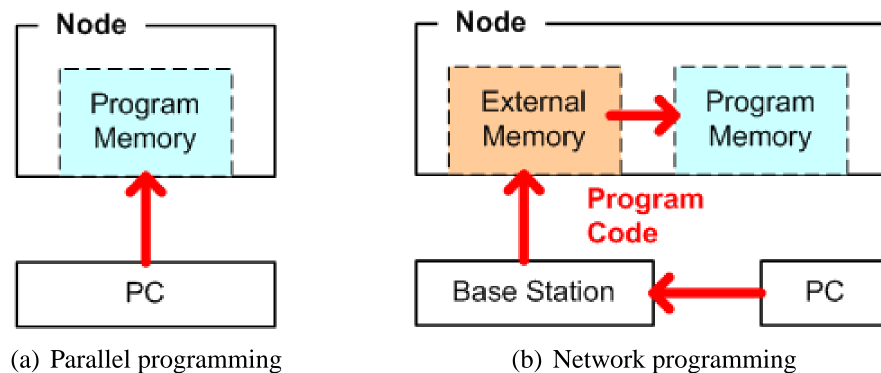
2.1 Network Programming

In this context, network programming means wirelessly installing a new program image on a sensor node. Instead of establishing a direct connection to the node and loading the program onto it, what is referred to as **parallel programming**, the binary image is sent in packets over the radio and stored outside the program memory. With this **network programming** approach, the downloaded code is then transferred to the program memory by the node itself. Figure 2.1 illustrates both parallel and network programming.

2.1.1 XNP

One of the very first approaches used to reprogram sensor nodes was included in TinyOS version 1.1. With **XNP** [CT03], mica2 and mica2dot nodes can be reprogrammed over the air. Only complete images can be transferred to the node, since XNP does not consider identical code parts. There is no forwarding mechanism in the program, so only the nodes in the immediate neighborhood of the basis station can be reprogrammed. This is also called **single-hop** reprogramming.

In order for the process to work, like in all other approaches, the reprogramming module has



(a) Parallel programming (b) Network programming

Figure 2.1: Network programming

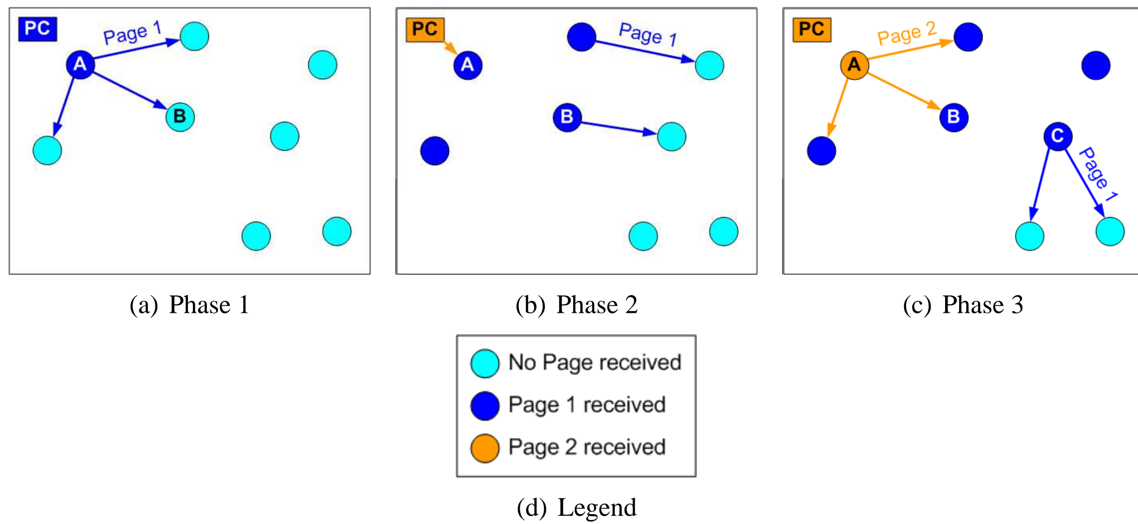
to be part of the application running on the node. Furthermore, a boot loader must be residing in the sensor node's boot loader section. It is usually installed on the node together with the application itself. A Java program splits up the new program code in packets and sends them to the node over the radio. The received packets are stored in the external flash memory by the sensor nodes, because it is not possible to write them directly into the program memory with the application still running. Another reason to write them to the external memory is the size of the image, because it does probably not fit into RAM. When all the packets have been sent, the node has the possibility to request missing or corrupted packets, until the complete image has been received. Then the program image is verified and the boot loader is called, which transfers the program code to the program memory. The system is then restarted, hopefully running the new application.

2.1.2 Deluge

Currently the "standard" application for wireless node reprogramming is **Deluge** [HC04]. It is integrated into the TinyOS system, and in contrast to XNP able to disseminate whole code images from one or more nodes to all the other nodes of a wireless **multi-hop** network. On each node, multiple application images can be stored in designated slots on the external flash memory, and it is possible to switch between these images. The data is represented as a set of fixed-sized pages that provide a manageable unit of buffer management and transmission. Every code image is tagged with a version number and the number of pages it consists of. Additionally, the node keeps a vector with the version numbers of each page of the current image, describing when it was updated the last time. This information builds a profile of the sensor node.

For the dissemination, an epidemic approach comes to use, as the network topology is changing constantly, with nodes entering or leaving unpredictably. To maintain a consistent state for the network, an adaption of the Trickle [LPCS04] algorithm is applied that is also referred to as "polite gossiping". Every node broadcasts "advertisements" periodically, announcing the version number of the images stored on it. Upon reception of an advertisement, the sensor node

checks if this image is already contained in its profile. If not, the new image is set up and a request message broadcasted. Every time a node realizes it is missing some or all pages of a newer version of an image, it requests the missing pages from the other nodes. While an image gets updated, only one page, split into packets, is sent and received at a time, to simplify the mapping of the data messages. As soon as a page has been received completely, the page is inserted into the profile of the node, increasing the number of potential senders of that particular page. This "spatial multiplexing" speeds up the distribution of the pages in the network by allowing pipelining of the transmission. The mechanism is illustrated in Figure 2.2.



(a) Phase 1 (b) Phase 2 (c) Phase 3

(d) Legend

Figure 2.2: Spatial multiplexing in Deluge.

The blue node in Figure 2.2(a), node A, is the source node. This node gets the image injected from a computer or PDA, here depicted as PC. Node A initiates the data transfer by broadcasting the first page to the other nodes in its neighborhood. All nodes that have completely received the first page turn blue. After having sent page 1 to the nodes in its neighborhood, node A starts receiving the second page from the attached PC in phase 2, shown in Figure 2.2(b).

Having completely received page 1, node B immediately advertises its newly received page to notify the nodes in its neighborhood of the new data. Theses node make a request then, to which node B responds by broadcasting the data. Node A delays its next advertisement to node B since it is currently receiving page 2 from the PC.

In phase 3, shown in Figure 2.2(c), node B is now ready to receive the next page from node A, page 2. At the same time, node C can propagate its newly received page 1.

Once a new image has been received, the system checks if it corresponds to the currently running image and initiates a reboot immediately. Otherwise, a reboot command can be sent to the node at any time, switching to a new application image. In order for the reboot to work, a **boot loader** has to be installed on the node. After every reset or reprogramming of a node, the boot loader is executed first, checking the external memory if a new image has to be loaded into program memory. Without the presence of such data, the boot loader does not take any action and just begins the execution of the current image. Otherwise, the selected image is copied from external memory into program memory and the new application is executed. Since TinyOS 1.14 [Web05], the TOSBoot boot loader is now installed by default when installing any TinyOS application.

2.2 Differential Network Programming

The drawback of the approaches presented above is the large amount of time they take to update large networks. Small bug fixes or updates for software maintenance do not really need all the code to be exchanged, often they affect only single lines of code. This holds even more for test cycles, when some new lines of code are inserted into a program an the developer would like to test it on a network where all the nodes have the previous test version of that program installed. Here, the differential idea comes into play.

2.2.1 Reijers and Langendoen

The first paper to discuss the advantages of differential updates was written by Reijers and Langendoen [RL03], presenting the idea only to distribute the changes to the currently running code. The differences between the old and the new code are summarized in an edit script of commands, that is to be processed by the nodes to rebuild the new code image. This diff script uses two basic commands: `insert` that appends a specified array of bytes to the code at the given position, and `copy`, that takes the sequence of bytes to be inserted at the current position from the given address. That way, the fact that sequences of code repeat or are similar in both codes is exploited. An additional command has been added, since there could be some sequences of bytes that only differ in very few positions. Instead of specifying a chain of `copy` and `insert` commands, a `repair` command can be used. This one corresponds to `copy`, but additionally corrects single bytes at specified offsets.

When inserting or deleting code at a certain position in memory, the following code is accordingly shifted up or down. This code shift changes the addresses of functions and variables which pose a problem. All the references in the code that follow after the inserted code have to be checked and possibly corrected. An example illustrating the problem is given in Figure 2.3. A group of functions has been moved from address `k` to address `m`, because new code was inserted. This means that each call to `f` will be a call instruction to address `m` instead of address `k` in the new binary, and the corresponding instruction will have to be repaired in the edit script. Reijers and Langendoen conducted some experimental data to find the percentage of instructions that are affected by address shifts. Figure 2.4 shows that about every sixth instruction would have to be a `repair` instruction.

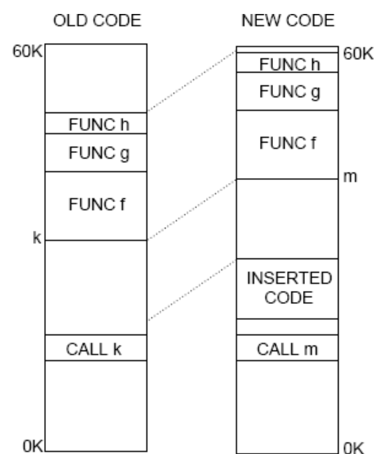


Figure 2.3: Address shifts due to inserted code [RL03].

Instruction type	Count	Percentage
call instructions	475	7.1%
RAM access	422	6.3%
mov instructions	291	4.3%
RAM as source	141	2.1%
RAM to register	99	1.5%
RAM as destination	159	2.4%
register to RAM	65	0.96%
mov constant to register	193	2.9%
constant value is an address	158	2.4%
push constant onto stack	18	0.27%
constant value is an address	18	0.27%
Total number of instructions	6707	100%
Vulnerable to address shifts	1073	16%

Figure 2.4: Number of instructions affected by address shifts [RL03].

This inevitably leads to a long edit script, even for minor changes. A special *patch list* command has been designed to adapt all addresses in a certain range with a specified offset. It is used for example after a `CALL` reference to correct the addresses to jump to. Even with this approach, some data has to be reset with `repair` commands if it was wrongly taken for an address.

2.2.2 Multi-hop Over-The-Air-Programming (MOAP)

Stathopoulos et al. [SHE03] extend and refine the ideas of Reijers and Langendoen. Their Multi-hop Over-The-Air-Programming (**MOAP**) application uses the same basic set of commands for an edit script, but add some special `copy` commands. The script is computed separately for both the code and the data part of the object file, and merged afterwards. Some `copy` commands can be optimized that way.

For dissemination, an algorithm called **Ripple** is used, that distributes the code packets to a selective number of nodes, not flooding the network. Corrupted or missing packets are retransmitted using a sliding window protocol, that allows the node to process or forward received packets while waiting for the retransmission of the missing packet.

2.2.3 Jeong and Culler

Another version of an update algorithm is presented from Jeong and Culler [Jeo05] with their **Incremental Network Programming** protocol. It relies on the Rsync algorithm [Tri99] for computing the difference files. This algorithm was originally developed to synchronize remote files over a network. It chops the data into smaller parts and computes their checksums to find variable-sized blocks that exist in both code images. Only blocks with different checksums are transmitted. In contrast to the original algorithm, the comparing is performed on the central computer, not requiring the node to do hardware intensive computation. A more detailed explanation of fixed-size block comparison and Rsync can be found in Section 3.3.2.

The complex `repair` command used by other algorithms is completely removed. While this command helps to minimize the size of the transmitted code, it also slows down the decoding process by requiring more time-intensive accesses to the external flash memory. The

characteristics of the Rsync algorithm lead to a big dependency of the performance on the type of difference of the codes. The Rsync algorithm works well for minor changes, but there are only very few identical blocks that can be copied in the code after major changes.

2.2.4 Koshy and Pandey

Koshy and Pandey [KP05] also use the incremental network programming scheme and try to cope with the code shift problem by using another concept they call **Remote Incremental Linking**. They realized that even for small changes, the code shift results in large difference protocols for the following code. Their approach starts at the linking stage, trying to take into account the possible consequences of a code shift. They developed a special version of the linker, that tries to keep the original memory layout of the code after changes. The linker places so called **slop spaces** in between the functions, allowing them to grow or shrink up to a certain extent, without having to shift the following functions. The determination of the size of these buffer spaces is everything else than trivial, a balance between wasting to much memory by choosing spaces too big, and having to adapt too much functions after a change for too small slop spaces has to be found.

In Figure 2.5, three functions `f`, `g` and `h` are laid out in memory. If `g` is updated and its size shrinks or grows, all the code below it is shifted. Function `h` remains the same, but still needs to be shifted, and all the pages indicated with tabs need to be rewritten.

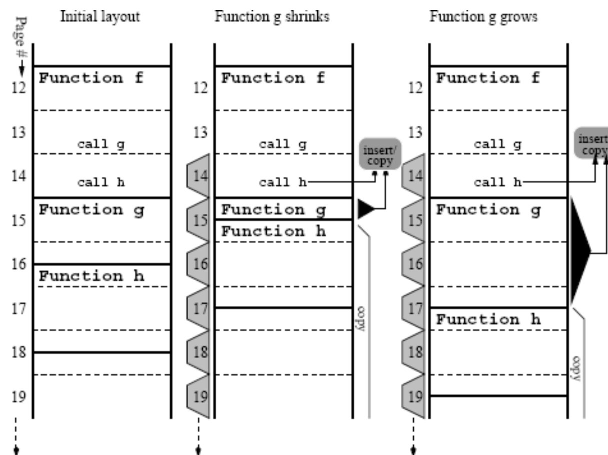


Figure 2.5: Memory layout for diff-based approach. Tabs indicate which pages are rewritten [KP05].

Figure 2.6 shows the approach taken by Koshy and Pandey. Functions are provided with a slop region to grow without running into other functions. As can be seen, the number of pages that has to be rewritten for the two scenarios where the function shrinks or grows has been reduced substantially. On the other hand, unused gaps in the memory layout are introduced. Of course this method only works if the slop space allocated is large enough, otherwise the same relocation as in Figure 2.5 takes place.

A special memory manager is added to the original linker, that tries to position the functions optimally in memory. The base station maintains hash tables for symbol resolution and linking related activities. Besides the large memory overhead, another problem arises after several

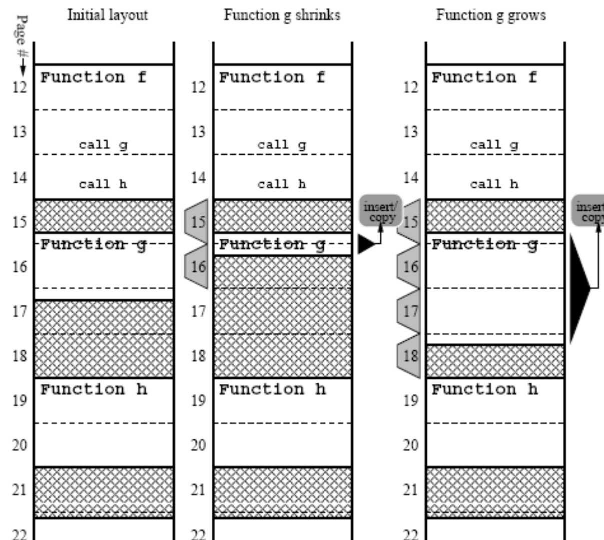


Figure 2.6: Memory layout for slop space approach. Slop space is hatched, and tabs indicate which pages are rewritten [KP05].

update processes, as the memory gets fragmented every time, what may require a complete reinstallation of the image.

A new algorithm has been chosen for the comparison and encoding of the differences. Xdelta [Mac00] is a binary diff algorithm, that produces difference files, also referred to as deltas. This algorithm is further described in Section 3.3.3. Unfortunately, an implementation of this paper is not available.

2.2.5 FlexCup

FlexCup [MGL+06] is another application used for differential reprogramming. Before the installation, FlexCup extracts the binary code for the TinyOS program and generates single components, that can be compiled independently, and are combined through clearly defined interfaces. The linking process takes place on the node itself. With these binary components, a global optimization of the code is no longer possible, it is restricted to the single components.

To be able to perform the linking on the node, some additional data is required. When an application is installed, the following meta-data is generated: some generic program information that describes the wiring of the components, a program-wide symbol table as well as a relocation table for each binary component in the program. It is also stored on the sensor node, using the external flash memory.

The exchange of a component works as follows: the binary code of the object files is extracted from the component to be exchanged on the central computer, including symbol and relocation table. Then the program code of the new component is transmitted to the node, as well as the new meta data. The nodes store the update information in the external memory. When the complete component has been received, it has to be linked on the node.

The linking at runtime requires several steps. First, the symbol table has to be merged with the newly received symbol data. After this task, for which a buffer of 3kB in RAM is needed, the symbol table is written back to the external memory. Then the relocation table has to be replaced with the new one. All the references in the relocation table have to be checked, whether they

need to be updated. Updates are required for references from the new component code and for references that changed their destination address. In the final step, the code of the new component has to be loaded into program memory by the FlexCup boot loader.

2.2.6 Dressler and Truchat

Dressler and Truchat [FTD06] present a method for distributed software management in wireless sensor networks using profiling techniques. They propose mobile robots, that perform management and configuration tasks in stationary networks. Software architectures can be maintained and applied for task allocation, sensor calibration, and general-purpose reconfiguration of surrounding sensor nodes, all based on the available resources at the robot systems. No new concepts for the network programming are presented, the exchange of software modules is built upon an extended version of Deluge.

2.3 Dissemination in Wireless Sensor Networks

Besides the discussed approaches dealing with the aspects of the update process, there is also some work that takes a closer look at the dissemination step.

A naive scheme of single retransmission of packets results in the broadcast storm problem [TNCS02]. This problem occurs when each broadcasted message prompts a receiving node to respond by broadcasting its own messages. These message in turn prompt further responses, and so on. The collisions and contention affects the reliability of the communication and also the energy efficiency. The following references focus on distribution of data in a wireless sensor network.

2.3.1 Trickle

Trickle [LPCS04] is the epidemic algorithm used by Deluge for propagating and maintaining code updates in wireless sensor networks. A “polite gossip” policy is applied, where nodes periodically broadcast a code summary to the local neighbors, but stay quiet if they have recently heard a summary identical to theirs. A node that hears an older summary than its own broadcasts an update. Instead of flooding the network with packets, the algorithm controls the send rate so each node hears a small trickle of packets, just enough to stay up to date. An implementation of Trickle is contained in TinyOS 2.x.

2.3.2 Sprinkler

Another algorithm for reliable data dissemination is Sprinkler [NASZ05]. It embeds a virtual grid over the network, whereby it can locally compute a connected dominating set of the nodes. Redundant transmissions can be avoided and a transmission schedule helps to avoid collisions.

2.3.3 Infuse

Infuse [Aru04] uses TDMA based scheduling to avoid collisions. Although TDMA guarantees collision-freedom during message communication, messages can be lost due to unexpected

channel errors (e.g., message corruption, environmental effects on signal strengths). To overcome such errors, implicit acknowledgments are used to recover from lost messages.

2.3.4 MNP

Multihop Network Reprogramming (MNP) [Wan04] is another reprogramming service for sensor networks. A Ripple-like propagation mechanism distributes the new image to the entire network. At each ripple, a subset of nodes is acting as source nodes, and all the other neighboring nodes are receivers. A publish-subscribe mechanism is used to prevent multiple nodes from becoming source nodes in the same neighborhood. Source nodes publish their newest version and all interested nodes subscribe. MNP suppresses the number of senders by selecting a few senders in a local manner. However, it does not pipeline packet forwarding and requires the entire image to be reliably received before the next ripple can begin. This results in larger latency. To optimize energy efficiency, the active radio time of a sensor node is reduced by putting the node into “sleep” state when its neighbors are transmitting a segment that is not of interest.

2.3.5 Gappa

To reprogram a sensor network, one can either communicate the entire new program to one (or a few) node in the field, or communicate parts of the code to a subset of sensor nodes on multiple channels at once. In the latter approach, the nodes need to communicate with each other to receive the remaining segments.

A protocol for such gossip between nodes is presented with Gappa [WK06]. To better utilize the multichannel resources and reduce contention, the protocol provides a multi-channel sender selection algorithm. This algorithm attempts to ensure that in any neighborhood, at any time, there is at most one sensor node transmitting on a given frequency. Moreover, the sender selection algorithm is greedy in that it tries to select the sender that is expected to have the most impact for each channel. The protocol also conserves energy by putting the nodes to “sleep” state that are unlikely to contribute or receive data shortly.

2.3.6 Aqueduct

While Deluge focuses on propagating the same code image to a network of homogeneous sensor nodes, this extension of Deluge is adapted to deal with heterogeneous networks more efficiently. The goal of Aqueduct [Phi05] is to limit the reprogramming traffic to the nodes that are interested in the update, while still being able to bridge the gaps between regions of interested nodes that are filled with uninterested nodes.

It introduces two roles for the nodes in a network, **member nodes** that are interested in receiving updates and cache the received parts in their external memory for later execution and **forwarding nodes**, that just forward the data packets without caching. The forwarding nodes are passive and do not download updates. The basic principle behind Aqueduct is that member nodes exchange code through normal Deluge interaction, but when member nodes are separated by more than one hop of forwarding nodes, the forwarding nodes create a bridge by acting as proxies for member nodes on either side of the gap.

Figure 2.7 illustrates the concept. The nodes interested in the updates are depicted as circles, while the squares represent the bridging nodes. Figure 2.7 (a) shows the propagation of ad-

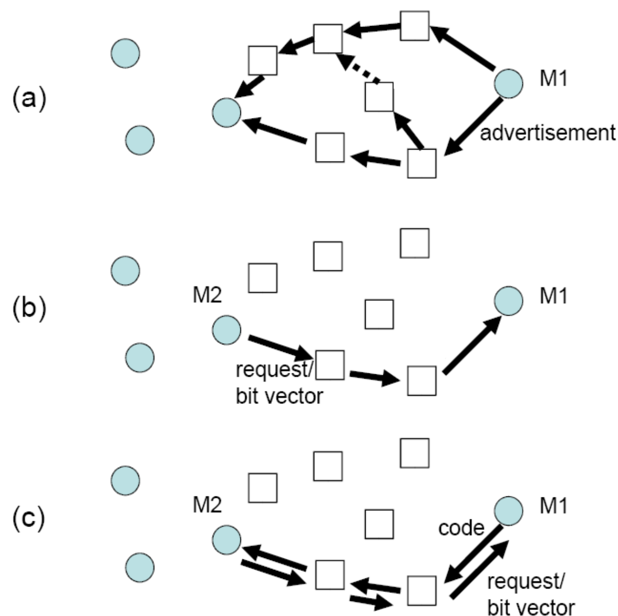


Figure 2.7: Forwarding bridge with Aqueduct in heterogeneous network [Phi05].

vertisements through intermediate nodes. These keep track of the advertisements received and the distance (in hops) to the source, and will reject identical advertisements that have a larger "distance to base" number than already observed. Just as in Deluge, the interested node M2 will unicast its request, containing a bit vector with the desired pages. In Deluge, this is unicast to an immediate neighbor, whereas in Aqueduct the request is unicast to the immediate parent closest to the advertising node. Figure 2.7 (b) shows that the request traverses the shortest path back to the member node M1. In the third phase, shown in Figure 2.7 (c), each node along the forwarding path pulls code image updates from its parent using the machinery of Deluge. Eventually, the code efficiently propagates to the destination M2 along this bridge, and does not involve any of the other forwarding nodes.

2.4 Virtual Machine Approach

2.4.1 Maté / Bombilla

Maté [LC02] is a compact virtual machine designed specifically for wireless sensor networks built on TinyOS. It has recently been renamed to Bombilla. Instead of installing applications as binary objects on the sensor node, every node executes a byte code interpreter. This interpreter reads the special byte code commands from memory, and transforms these Maté operations to TinyOS operations. Therefore, there does not have to be reinstallation and rebooting of an application since the program is just some input data for Maté. Maté also contains support for application code distribution, which is named code infection. Although virtual machines are promising as system software for wireless sensor networks, their space and energy overhead can render them counterproductive. Also, due to the restricted instruction set of Maté not all semantics that are possible by specifying the program in nesC can be expressed.

2.4.2 SensorWare

In SensorWare [BHS03], the developers set very high requirements on the hardware. It does not fit into the memory of popular sensor nodes and targets richer platforms to be developed in the future. In contrast to Maté, also complex semantics can be expressed. The program services are grouped into theme related APIs with Tcl-based scripts as the glue. Scripts located at various nodes use these services and collaborate with each other to orchestrate the data flow to assemble custom networking and signal processing behavior. Application evolution is facilitated through editing scripts and injecting them into the network. Both SensorWare and Maté are limited in that they support application updates only, by replacing high-level scripts. They do not permit the lower level binary code to be modified.

3

Analysis

In this chapter, we will first look at the requirements for the project in Section 3.1.We will decide on the basic utilities for general network programming in Section 3.2. Then, we discuss different algorithms for the encoding of the code to be transmitted in Section 3.3 and analyze several approaches for the recombination of the code in Section 3.5.

3.1 Requirements

The goal of this project is to allow the differential update of nodes in a wireless sensor network. This process consists of several steps. First, the nodes have to be **manually programmed** with the basic update software. This step requires a physical connection to the nodes and therefore has to be done before the nodes are distributed over the application area. Then, an application can be **wirelessly installed** on the nodes, which requires the complete code of that application to be sent to each node. If we want to reprogram the node, the code of the old and the new application is compared and a difference file is computed with an **encoding** algorithm. This encoding step takes place on a computer, where no special restrictions regarding computational power or energy usage are imposed. The focus here is on the size of the difference file, which should be as small as possible, to save time and energy when sending it to the nodes.

This difference file, together with information about the application that should be updated on the node, is then distributed to all nodes in the networks in the **update dissemination** step. The file is split into small packages that are stored in the external flash memory by the nodes. All nodes have to be reached eventually trying to save time and energy. When all nodes have received the update, no further communication between the nodes is necessary. We require the nodes to send an **acknowledgment** after successful reception of the update files.

After having completely received the difference file, the nodes start the **decoding** process. The difference file is processed to build the code of the new application, involving also the code

of the current application. An important restriction here is the impossibility to dynamically allocate memory in a TinyOS program. Hence, no dynamic data structures such as hash tables can be used. This is a severe restriction, since a lot of compression programs require such data structures. As soon as the application image is built, the node initiates a restart and the new image is transferred to program memory. This step requires a **boot loader** that performs the transfer on a restart before the current application starts its execution. Finally, we would like the reprogrammed nodes to send a **completion feedback**, to be informed about the successful restart.

3.2 Wireless Dissemination

To wirelessly install an application on the sensor node, one of the approaches discussed in Section 2.1 has to be applied. We want to be able to send complete code images from one node to another. A data structure has to be developed to manage the code images, and to split them up into pages and packets for transmission. The transmission protocol has to be suited for large objects and must be reliable since low bandwidth and high loss rates are typical in wireless sensor networks. We choose to rely on the implementation of Deluge and use its basic functionality to send complete application images to a sensor node. Deluge also allows to store multiple application images in the external memory, making it possible to switch between these.

3.3 Encoding

The code patch to be disseminated in the network has to be put into a certain format in order for the nodes to decode it and build the new image. Since we know the original file (source) as well as the new version of the file (target), we should use this information during encoding. The process of computing a “patch” of minimal size between two files is called **delta encoding** or differential compression. There are several techniques for computing such a delta. The basic problem goes back to the string-to-string correction problem [WF74], the task to find a minimal edit script that converts a reference string into a target string. The operations used are `insert` and `delete`. However, these algorithms, also referred to as *insert-delete* algorithms, do not take into account that the data common to both files may not appear in the same order in the two files. Neither does is capture substrings that are recurring several times.

The framework has been extended to the string-to-string correction problem with block moves by Tichy [Tic84]. In contrast to the *insert-delete* approach taken before, a sequence of `copy` and `insert` operations is used to describe the changes. Such algorithms are described as `copy-based` algorithms. The basic idea is to use pointers to substrings in the source file to construct the target file. A further extension was presented with the well known *Lempel-Ziv* compression algorithm [ZL77], which compresses a string by substituting its prefix with a reference to already compressed data.

Delta algorithms are applied for example in software revision control systems where multiple versions of each code file have to be stored. Changes between subsequent versions are typically small, so substantial amounts of disk space can be saved by storing their difference information. It has also been proposed to use deltas for improving HTTP performance by sending the difference for outdated web pages.

We will now compare some of these delta algorithms. A theoretical overview on differential compression is given in [ABF+00], while delta algorithms are further described and analyzed in [SM02] and [HVT96].

3.3.1 Diff

The most popular and widely used delta algorithm program is the file comparison utility Unix **diff** [HM76]. The program finds an approximation of the longest common subsequence for whole lines of text. By just considering changes on a per-line level, it is much faster than other algorithms that compute differences on a byte-level. But with that restriction, the size of the produced deltas is not optimal. Furthermore, it can only be used for text files.

3.3.2 Rsync

The Rsync algorithm [Tri99] was developed to synchronize files and directories from one location to another, while minimizing data transfer using delta encoding when appropriate. This copy-insert algorithm divides files into blocks and decides for each block if it has to be downloaded from the remote location or if it is identical. In our application scenario, we do the compression and decompression locally, so downloading can be replaced by a simple insert of a data sequence from the difference file.

The source file is partitioned into equal blocks of a certain size. For each block, two checksums are computed: a reliable but expensive, and an unreliable but fast checksum. Now, we iterate over all positions in the target file and consider the block that follows after that position. We use the same size for this block as we used for the source file. For every block, we compute the unreliable checksum. As we always only shift the block boundaries by one byte, the checksum can be computed very efficiently by using a 32 bit "rolling checksum". Its computation can be done in constant time. We now compare the hash value with the values of the source file blocks, and if a matching block is found, we also compute the reliable checksum. The reliable checksum used is a MD4 128 bit hash function. If both checksums match, a pointer to the index of the matching block in the source file is stored in the diff file, and the position in the target file is advanced by the length of the match. If no match occurred, the symbol at this position is added to the diff file and the position advanced by one. The decoding is straightforward, as the patch only consists of copy and insert instructions. Pointers to data blocks in the source file have to be copied, and single bytes or byte sequences have to be inserted at the current position in the target file that is being reconstructed.

The choice of a good block size is critical for the performance of the algorithm. If the two files to compare are very similar, a large block size is more efficient. On the other hand, if we have two files with a lot of differences, a small block size allows more matching blocks to be found. Another important role plays the distribution of the differences. If they are equally distributed over the whole file, Rsync may be completely inefficient, since there will be a difference in each block. In practice, the block size is adapted during the encoding with certain heuristics.

3.3.3 Xdelta

Xdelta [Mac00] is another variant of a copy/insert delta algorithm. Contrary to Rsync, it does not compare single blocks with each other, which makes sense for remote file synchronization,

but it compares every single byte of the two files. The difference file consists of pointers to data sequences in the source file, and also to the already decoded part of the file being reconstructed. This further optimizes the size of the difference patch. The data sequences to be copied can be of arbitrary size, whereas the encoder tries to maximize the length of these sequences. If no matching data blocks can be found in the source or target file, the `insert` command is used. An additional `run` command is provided, that expresses a sequence of the same byte of a certain length.

```
computeDelta(src, tgt)
  i ← 0
  sindex ← initMatch(src)                      ▷ Initialize string matching.
  while(i < size(tgt))                         ▷ Loop over target offsets.
    (o, l) ← findMatch(src, sindex, tgt, i)    ▷ Find longest match.
    if(l < s)
      outputInst({insert tgt[i] })             ▷ Insert instruction.
    else
      outputInst({copy o l })                  ▷ Copy instruction.
    i ← i + 1

initMatch(src)
  i ← 0
  sindex ← empty                               ▷ Initialize output array (hash table).
  while(i + s ≤ size(src))                     ▷ Loop over source blocks.
    f ← adler32(src, i, i + s)                 ▷ Compute fingerprint.
    sindex[hash(f)] ← i                        ▷ Enter in table.
    i ← i + s
  return(sindex)

findMatch(src, sindex, tgt, o_tgt)
  f ← adler32(tgt, o_tgt, o_tgt + s)           ▷ Compute fingerprint.
  if(sindex[hash(f)] = nil)
    return(−1, −1)                             ▷ No match found.
  o_src ← sindex[hash(f)]
  l ← matchLength(tgt, o_tgt, src, o_src)      ▷ Compute match length.
  return(o_src, l)
```

Figure 3.1: Pseudo-code for the Xdelta algorithm [Mac00].

Figure 3.1 shows the pseudo-code for the Xdelta algorithm. The main function is `computeDelta`, that takes the source and the target file as inputs. It builds a string matching data structure for the source file by calling the `initMatch` function. That function computes a hash table of fingerprints for source blocks of a certain length. If hash collisions occur, the hash for a block that appears first in the source file will always overwrite the other. The reason that fingerprints for the earlier blocks are preferred is, that they potentially lead to longer matches. The `findMatch` function then performs the string matching with the target file. The target file is split into fixed-size pages, that are searched for matching fingerprints in the source hash table. When a match is found, it checks the source file with a direct string comparison, and tries to extend the match in both directions in the `matchLength` function.

3.3.4 Zdelta

The Zdelta algorithm [TMS] uses the same basic ideas as Xdelta. The main difference is that Zdelta also encodes the computed delta. The zlib compression library [Gai] is used to further compress the produced delta with Huffman codes. Although it produces slightly smaller deltas we can not use this algorithm. For the decompression of the delta, dynamic hash tables are required. On the sensor nodes, dynamic memory allocation is not possible, therefore this algorithm is eliminated from our choice.

3.3.5 VCDIFF

VCDIFF [KMMV02] is a general and portable data format for encoding compressed data or differencing data, so that it can be easily transported among computers. This format is also used in the Xdelta3 implementation. The decoding algorithm is independent from string matching and windowing algorithms. This allows free choice of the encoder while keeping the same decoder. The decoding time is proportional to the size of the target file, and uses space proportional to the maximal window size.

The target file is partitioned into fixed-size blocks, called windows, which are processed separately. Each target window can be compared against another window that is either part of the provided source file or part of the already encoded target file. VCDIFF also uses the typical *copy-insert* instructions. The instructions to encode and direct the reconstruction of a target window are called delta instructions. Besides `copy` and `add` (insert), there is an additional `run` instruction, that is used to repeat a certain byte several times.

Below, there is a simple example taken from [KMMV02] to illustrate the source and target windows and the delta instructions.

```
a b c d e f g h i j k l m n o p
a b c d w x y z e f g h e f g h e f g h e f g h z z z z

COPY  4, 0
ADD   4, w x y z
COPY  4, 4
COPY 12, 24
RUN   4, z
```

The upper string represents the source window, while the lower string corresponds to the target window. That is, the part of the target file that is currently being decoded. The two windows are concatenated, so that the first letter `a` in the second line is at location 16. The first `COPY` instruction tells the decoder to copy four letters from the beginning of the source window. Then, we insert the four letters `w x y z` at the current position (which is 20). Another four letters are copied from the source window. Now, the following 12 letters are not copied from the source file, but from the currently decoded target window. Address 24 corresponds to position 8 in the target window. The data to copy from overlaps with the data to be copied, what is fine as long as the source copy address starts before the destination address. That way, periodic sequences can be encoded efficiently. The last `RUN` instruction, that appends 4 times the letter `z`, is a compact way to encode a sequence repeating the same byte.

We chose this algorithm for our implementation for several reasons. It is a very efficient algorithm and the size of the generated delta files is small enough. It outperforms Rsync in the quality of compression. Zdelta that offers even better results can not be used because of the impossibility of dynamic memory allocation. An implementation for Xdelta is available for Linux, so we can use this program for the encoding process.

3.4 Update Dissemination and Feedback

To distribute the delta files over the network, we use the mechanisms already present in Deluge. The epidemic dissemination guarantees fast distribution, but uses a lot of bandwidth due to the large number of messages exchanged. Every node broadcasts its profile periodically, and even though the number of these advertisements slows down after some time with no new data present, it causes substantial network traffic. To reduce some of this overhead, we only send advertisements during the update or installation process. Once an image has been installed or updated, the advertisements are stopped since we do not assume additional nodes that enter the network and have to be reprogrammed as well.

As soon as a node hears about an update or a new image to be programmed it stores the node address of the first sender of such an advertisement. The sender becomes the parent of that node in an **implicit feedback tree**. The node starts with sending request messages to its parent. When the data has been completely received from the parent node, a reception message is sent to the parent. To make this communication reliable, the node repeats the sending until it receives an ACK message. Any node that receives an update message, which always comes from its implicit child node, forwards this message to its parent node and sends an acknowledgment to the child node. The acknowledgment chain ends at the base station.

3.5 Code Recombination

Once the differential update file is stored at the node it has to be recombined to the new image and the code that is currently running has to be overwritten. There are three general options to do this which will be described in this section.

3.5.1 Halve Memory

For some applications, using half of the available memory may be sufficient. In this case, the memory is split in an upper and lower half, and, with the current code running in the upper half, the new code image is built in the lower one. Once the image is built, a small piece of code is placed in RAM, and executed to copy the bottom half to the top half and to reboot the node on completion. Since the processor is running code from RAM, there is no problem in overwriting the old code.

3.5.2 2 Phase Approach

The previous approach can be extended to use all available memory. We can split the code into two halves, and place all the critical code required for our code distribution scheme in the bottom half. The rest of the bottom and the whole top half can be used for application code.

In the initialization phase the application is stopped and the top memory half is cleared. The critical code in the bottom half keeps running. Then, the first approach is used to update the bottom half containing the critical code. Once this is done, the new critical code is used to build the new top half containing the rest of the application code. Only when this is completed we can restart the application.

This approach has a number of drawbacks. First, the application will be stopped during the whole process instead of just during the copying of the new image. Second, we need to do the verification step for both halves increasing the overhead of this approach. Finally, this approach is only possible if the critical code is smaller than half of the memory.

3.5.3 Build in EEPROM

In our implementation we use the external EEPROM memory as temporal buffer. We simply build the new code image in one of the EEPROM sectors and use a small piece of code in RAM that tells the boot loader to load the image into the program memory on a restart. An advantage of this approach is that once the image is built we can have multiple images stored in EEPROM and load them when necessary.

4

Design and Implementation

This chapter outlines the architecture of our implementation and describes the most important modules. The extension of Deluge is described in Section 4.1. The encoding of the delta file is explained in Section 4.2 and the implementation of the update mechanism in Section 4.3. The decoding process is described in Section 4.4.

4.1 Wireless Dissemination

For the dissemination and management of the application images the Deluge application has been adapted. Changes to the basic functionality of the original version include the disabling of periodic advertisement messages and the feedback mechanism. Advertisements are now only sent during the update process and stopped when the images or update files have been distributed throughout the network.

A feedback mechanism has been added, that allows status updates for every single node. A simple implicit node hierarchy is established while the updates are propagated. When a node receives an advertisement for an image that it does not yet know, it stores the sender of that advertisement as its parent in the implicit tree. As soon as the image or diff file has been completely received, a notification message is sent to that node. The sending is repeated, until the node receives an acknowledgment for the message. Any node that receives an update notification message forwards it to its parent node and sends an acknowledgment to the sender of the notification. The process repeats until all messages are acknowledged. For the case that the network bandwidth is too small and the update notification messages do not get acknowledged, the nodes give up after a certain number of tries.

To store the application images on the node, the external flash memory is divided into slots. Two additional slots are added to the normal slots used by Deluge, one for storing the delta file, and a second for the reconstruction of the new image. We will now go into more detail and have

a look at the format of the code images, as they are stored on the node, in Figure 4.1.

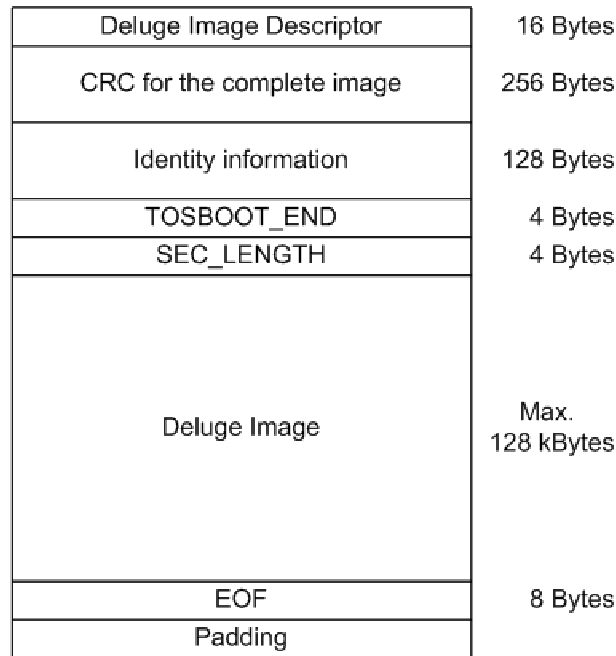


Figure 4.1: Memory layout of Deluge image.

At the lowest memory address is the **Deluge Image Descriptor**, a struct that stores information about the image. Its structure is shown in Figure 4.2. Then the Cyclic Redundancy Check

```
typedef struct DelugeImgDesc {
  uint32_t  uid;             // unique id of image
  imgvnum_t vNum;            // version num of image
  imgnum_t  imgNum;          // image number
  pgnum_t   numPgs;          // num pages of complete image
  uint16_t  crc;             // crc for vNum and numPgs
  uint8_t   numPgsComplete;  // numPgsComplete in image
  uint8_t   reserved;
} DelugeImgDesc;
```

Figure 4.2: Deluge Image Descriptor struct.

(**CRC**) data follows. It ensures the integrity of the complete image. The check is performed by the boot loader before transferring the image into program memory. After that, the **Deluge identifier** block follows. This information is presented to the user, when he investigates a node by sending a ping message. Two addresses are placed next, the address in memory where the boot loader TOSBOOT ends (**TOSBOOT_END**), and the length of the Deluge image (**SEC_LENGTH**). These two addresses are used by the boot loader to guarantee that the boot loader code does not get overwritten. The binary code for the complete **Deluge image** follows, terminated with an End-Of-File (**EOF**) delimiter. The space between this image and the beginning of the next one is padded out.

4.2 Encoder

The installation of the update is done in several steps. The first step is the computation of the delta file. The input to the encoder is the binary code of the two applications. We need to have exactly the same data that is already stored on the node, since the layout of the reconstructed image has to match as well. The binary code of an image can be found in its `tos_image.xml` file. When an image gets injected into a network with Deluge, the Deluge java tool chain program extracts the binary code from that file and adds some additional data like the CRC data. We do exactly the same for both the source and target file resulting in the same code layout as presented in Figure 4.1. The only difference is that we do not have the 16 Bytes Deluge image descriptor at the beginning.

Now we compare these two data structures beginning with the CRC code and ending with the image in order to generate a delta. The encoder we use is the current Xdelta implementation, Xdelta3 [Mac]. Our executable produces deltas in the VCDIFF format (see Section 3.3.5), without further compressing them. Unfortunately, the Xdelta3 code is only available for Linux. According to its developer, the code is being worked on to compile also on Windows machines in the future. With Linux, the encoder is integrated in the update program. As long as the encoder is not available under Windows, the delta files have to be produced by hand. First we have to generate the source and target file, by calling a helper function with the java tool chain:

```
java net.tinyos.tools.DelugeD -w
    -ti=build/tinynode/tos_image.xml -o=<source or target>
```

This step has to be done for the source as well as for the target application. The `-ti` parameter is the original `tos_image.xml` file, while the `-o` parameter is the filename of the result. These two generated files are the input for the encoder. The usage of the compressor is as follows:

```
xdelta3 -s <source> <target> > DELTA
```

4.3 Update Installation

To send a delta file to the nodes and initiate an update, an advertisement message is sent to the base node. The advertisement contains a Deluge image descriptor. The number of the image to be updated is given, as well as the version number of that image increased by one to announce a change. The number of pages in the struct however refers to the size of the delta file. The reserved field is used to indicate the update. The delta file is now received by the node and stored in the delta slot of the external flash memory. The usual integrity checks are omitted since the delta does not fit into the conventional image format. Another information sent with the advertisement is the slot number where the reconstructed image should be stored. This information is also added to the reserved field. When the complete delta has been received and the reception feedback has been sent the decoder is started. It processes the delta file and copies the needed data from the image slot where the source image is stored. The new image is constructed in the update slot of the external memory. Figure 4.3 shows the decoding process and the involved data. There are three image slots shown in the illustration. The middle one is the slot where the delta gets stored, and the right one is the “construction” slot for the update

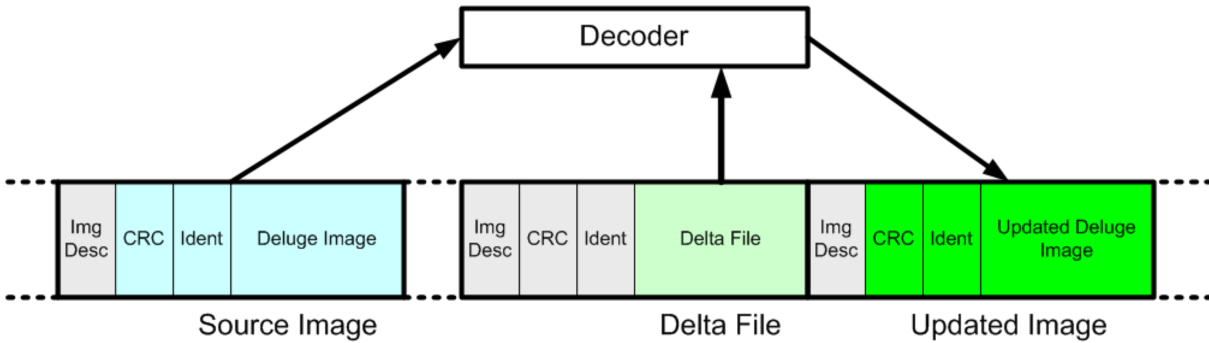


Figure 4.3: The data sections involved in the decoding of the delta file.

image. In the left slot lies the source image. The colors clarify which data is involved in the decoding process.

With the decoding process finished, the image is copied to its destination slot. If it corresponds to the currently running application, a flag is set and the node is rebooted. The boot loader transfers the code to program memory and starts its execution.

4.4 Decoder

The task of the decoder is to process the VCDIFF encoded delta file and construct the target image. The decoder is divided in two components: One for the handling of the delta file and another one to carry out memory accesses.

The delta decoder reads the delta instructions and executes them. The processing of a delta instruction is implemented as a nesC task which is posted by the Deluge main component. That way, other routines do not get blocked since the decoding may take a while. It is possible that the decoding starts on one node while another node has just received the delta and waits for its feedback message to be acknowledged

The execution of the delta instructions requires read and write accesses to EEPROM memory. A characteristic of EEPROM memory is that the number of rewrite cycles is limited. Changing a single byte is only possible by rewriting a whole block. In practice, this means a block has to be erased completely before it can be rewritten. These repeated write and erase cycles eventually damage the thin insulating layer, a process called 'wear out'. Typically, about 10'000 erase-write cycles are possible with an EEPROM module.

To access the EEPROM memory, the `PageEEPROM` component is used. This component is able to read and write whole pages. As a consequence of the restrictions outlined above, our implementation of the EEPROM storage module uses a buffer to keep the page that is currently worked on in RAM. By keeping this page granularity we try to reduce the number of read or write accesses to the memory. Successive reads to the same page can be done immediately and with only one EEPROM access as the page is already in RAM. When we access another page, the current buffer is only written to memory if it is dirty, that means changes were made to it. To write, the page has first to be read into the buffer, then the desired bytes are written in the buffer and before the buffer is written back to memory the page has to be erased. This is required because of the EEPROM write restricions described above. Due to the buffer, `write` commands are deferred until another page is accessed, or the `flush` command is invoked.

```
interface EEPROMStorage {
  command result_t init(startPage);
  command result_t read(offset, data, length);
  command result_t readUint32(offset, uint32, inc);
  command result_t write(offset, data, length);
  command result_t run(offset, byte, length);
  command result_t copy(sourceOffset, targetOffset, length);
  command result_t sync();
  command result_t flush();
}
```

Figure 4.4: The EEPROM Storage component.

Unfortunately, the decoder needs a more sophisticated interface to the memory. It does not only require commands to `read` and `write` single bytes or byte sequences but also a functionality to `copy` a sequence of bytes from one address in memory to another or to repeatedly write the same byte (`run`). The interface of the storage module is shown in Figure 4.4. The `readUint32` instruction is a convenience function that reads and decodes an unsigned 32 bit integer from the given memory address and stores the number of bytes used in memory for that integer (1 – 4 Bytes).

5

Results

This chapter discusses results achieved with our implementation. In Section 5.1 we evaluate the size of the delta patches and in Section 5.2 we analyze the time used for the update process.

5.1 Size of Code Updates

An important performance measure is the size of the delta patches used to update the applications. There are scenarios where the new image is almost identical to the current image, resulting in very small delta files, and other scenarios where we replace one application with a completely different application. In that case, the size of the delta depends on the quality of the compression. The number of bytes to send will be smaller in any case as there are always similar code sections in two different application images. We will look at some test examples in Table 5.1. We applied the Xdelta encoder to several application images. The example applications without integrated Deluge support (e.g. `Blink` or `CntToLeds`) have a code size that is a lot smaller than those with Deluge support (e.g. `BlinkDeluge`), but once they are loaded into program memory, they do not support any further network reprogramming. So the relevant example applications are those that integrate Deluge. The additional code size is about 30 kB. That is 8 kB more than the normal Deluge version, and it is mostly caused by the memory access and decoding functions.

As can be seen from the examples, the encoding ensures a small delta size for the similar applications like `BlinkDeluge` and `BlinkFastDeluge`. `BlinkFast` is the same application as `Blink`, the only difference lies in the blink frequency, so only a constant has been changed. The reason the delta is not only a few bytes in size is the delta header overhead. A few bytes are needed to indicate the window sizes and some additional information is included. In most application scenarios only some lines of code change, so the usage of our update tool will be similar to this case. As another example, we evaluated the delta size for an

Application	Initial binary size [bytes]	Final binary size [bytes]	Delta size [bytes]	Compression ratio
`Blink` to `BlinkFast`	3500	3500	79	2.3%
`Blink` to `CntToLeds`	3500	3494	984	28.2%
`BlinkDeluge` to `Blink-FastDeluge`	34482	34482	254	0.7%
`BlinkDeluge` to `CntTo-LedsDeluge`	34482	34456	7440	21.6%
`BlinkDeluge` to `Oscillo-scopeDeluge`	34482	38374	17789	46.4%
`CntToRfmDeluge` to `Cnt-ToLedsAndRfmDeluge`	34714	35012	7604	21.7%

Table 5.1: The code sizes of delta patches and original images for different update scenarios.

update of the `CntToRfm` to the `CntToRfmAndLeds` application, both with integrated Deluge support. We see that for applications that have different code, like `BlinkDeluge` and `OscilloscopeDeluge`, the size of the delta is still only half the size of the new application image.

5.2 Time Used for Updates

To test the performance of the update mechanism in terms of time, we used two test scenarios. The nodes have been arranged in star topology for the first test and in line topology for the second.

5.2.1 Star Topology

In the first scenario, the sensor nodes are placed in an environment where every node is able to reach all other nodes. This means that the maximum distance between any of the nodes does not exceed about 200 m, which is the maximum radio range of a TinyNode mote. This “star” setup is shown in Figure 5.1.

The results for this star topology for an update from `Blink` to `BlinkFast` are shown in Table 5.2. We have measured the time for the dissemination of the delta file first. A second measurement indicates the total time it takes until all the nodes have received and decoded the update. Finally, the total time of the process including the feedback messages is listed together with a comparison of the time used for the process by Deluge. All of the test applications have integrated differential Deluge support. For such small changes our solution outperforms Deluge. The amount of time saved by sending only the small delta exceeds the additional time we invest for the decoding on the node. With 10 nodes, the delta is disseminated to the nodes in 23 seconds and the update is complete after 26 seconds. If we perform the update with the

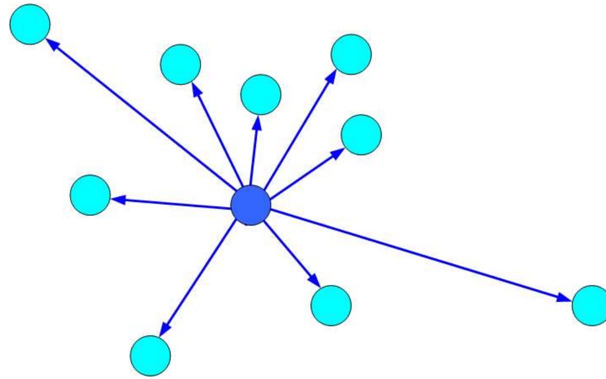


Figure 5.1: Test setup with the nodes arranged in the star topology.

Blink to *BlinkFast*				
Number of Nodes	Time for Dissemination	Total time for Update	Total time with Feedback	Deluge total time
3	8 s	22 s	24 s	126 s
10	10 s	23 s	26 s	131 s

Table 5.2: Update time for a Blink to BlinkFast update in the star scenario.

normal Deluge version, the application is smaller, but the process still takes about 5 times longer as all the data has to be sent.

The results for an update between two different small applications like `Blink` and `CntToLeds` are shown in Table 5.3. The differential and the normal version of Deluge take about the same time for the process. The time saved by only sending 20 percent of the amount of data to the node is spent on decoding. It only takes 34 seconds to disseminate the update, but another 95 seconds to process the delta. The number of nodes does not heavily influence the performance of both update tools. A slight advantage results for the differential approach. With less data to transmit, also less retransmissions of packets have to be done.

A worse scenario for the differential Deluge is shown in Table 5.4. As `Oscilloscope` and `Blink` do not have much common code sections, the patch saves only half the size of the data transmission. The decoding is very time intensive here, which is the reason the differential approach takes longer than just sending the complete new image.

Blink to *CntToLeds*				
Number of Nodes	Time for Dissemination	Total time for Update	Total time with Feedback	Deluge total time
3	34 s	125 s	128 s	126 s
10	34 s	128 s	140 s	130 s

Table 5.3: Update time for a Blink to CntToLeds update in the star scenario.

Blink to *Oscilloscope*				
Number of Nodes	Time for Dissemination	Total time for Update	Total time with Feedback	Deluge total time
3	67 s	224 s	228 s	165 s
10	85 s	240 s	246 s	172 s

Table 5.4: Update time for a Blink to Oscilloscope update in the star scenario.

Blink to *BlinkFast*			
Number of Nodes	Time for Dissemination	Total time for Update	Total time with Feedback
3	17 s	31 s	37 s
10	72 s	85 s	112 s

Table 5.5: Update time for a Blink to BlinkFast update in the line scenario.

5.2.2 Line Topology

In the second test scenario we assume a distribution of the nodes in the network that corresponds to a "line". Each TinyNode is placed in a line with the maximum radio range distance between them. That way, the reprogramming process will take place in a multi-hop fashion as the pages of the update have to be sent from one node to the other, only taking one hop at a time. Figure 5.2 shows the line topology. The dissemination of the update takes longer that way, but thanks to page pipelining the nodes do not have to wait for the complete update and can start forwarding pages as soon as they have received the first page.

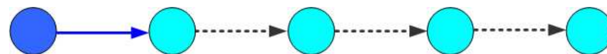


Figure 5.2: Test setup with the nodes arranged in line topology.

In Table 5.5, we show the results for the line topology for the `Blink` and `BlinkFast` applications. Compared to the star topology, the time for the dissemination does not depend so much on the number of pages to be sent, but on the number of nodes in the line. The time for the decoding remains the same, while the feedback takes longer.

We also conducted the measurements for the `Blink` and `CntToLeds` applications, which are presented in Table 5.6. Once again, the update takes more time to reach all nodes in the network. However, as the amount of data to send is bigger than in the previous example, the delay caused through this topology becomes less important for the overall performance.

Blink to *CntToLeds*			
Number of Nodes	Time for Dissemination	Total time for Update	Total time with Feedback
3	71 s	164 s	169 s
10	124 s	218 s	237 s

Table 5.6: Update time for a Blink to CntToLeds update in the line scenario.

6

Conclusions and Future Work

In this chapter, we draw a conclusion and list some possible extensions to the existing system.

6.1 Summary

In this master thesis, an application to update wireless sensor networks with differential patches was designed and implemented for TinyOS on the TinyNode platform. The Xdelta algorithm was chosen for the differential compression. The delta patches to be disseminated in the network are generated in the VCDIFF format by the Xdelta3 encoder. To be informed about the update process, the Deluge application was extended with feedback functionality. The system has been tested in several scenarios, showing a good performance for updates with applications that are similar.

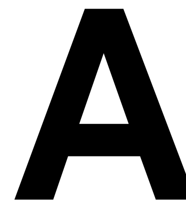
6.2 Evaluation and Discussion

Chapter 5 shows good results for updates with similar code images. Our solution is targeted to facilitate debugging and testing. Normally, the developer has to test an application, fix some bugs or add a certain functionality and test it again. During these test cycles, the amount of code changed in the application is usually small, so our solution is suited to speed up the development process. As soon as we have larger sections of different code in the two versions, the time used for the decoding of the delta patch grows bigger. The advantage of having less data packets to send is degraded. If we completely replace an application with a different one, the results of our solution are similar to the Deluge implementation, or even worse. For better results, the EEPROM access would have to be optimized for the decoding component.

6.3 Future Work

There are a few things which could improve the performance of the current solution. The encoder should be completely integrated into the tool also for Windows environments. This can be achieved as soon as the Xdelta3 project is also available under Windows. Through personal communication with the developers of the Xdelta3 project, they are currently working on porting the code. The integration is then straightforward as only one additional call of the executable by Java's runtime environment is sufficient.

The feedback could be improved to support a visual representation of the sensor nodes in the network. The update application could be integrated in the controlling tool which has also been developed at the Distributed Computing Group [Cad06]. In that thesis, a framework was designed, that allows to watch and inspect components remotely. An Eclipse plugin was developed to keep an overview of the network.



Software Manual

The following tutorial on the differential Deluge implementation is based on the Deluge tutorial [Ber05]. Only the new functions and the differences to the Deluge tutorial are described below.

A.1 Installing the Boot Loader

DelugeDiff requires a boot loader, TOSBoot, to reprogram the node. To ensure that TOSBoot installs correctly when installing a TinyOS application we try building and installing Blink on a TinyOS node by going to `tinyos-1.x/apps/Blink` and typing the following:

```
% make tinynode bsl.<x> install.1 TINYOS_NP=BNP
```

where the `x` in the `bsl.x` parameter is the number of the COM port we use to connect to the node. If we work with TinyNode we have to decrease the COM port number by one. So if we connect a node over port COM 3, we have to provide `bsl.2` as parameter.

The most important parameter is `TINYOS_NP=BNP`. For other environments, TOSBoot is installed with every application by default since TinyOS version 1.1.15 [Web05]. Unfortunately, this is not true for the TinyNode environment. This is why we have to add the `TINYOS_NP` parameter to install the boot loader manually. A problem may arise if the boot loader has not yet been compiled or if its path can not be resolved. We have to check the `tinynode.target` file, which can be found in the shockfish contribution folder under `tinyos-1.x\contrib\shockfish\tools\make\tinynode.target`. Here, we may have to change the following line:

```
BOOTLOADER := $(shell cygpath -m $(BOOTLOADER))
```

by adding quotes to the path name as follows:

```
BOOTLOADER := "$(shell cygpath -m $(BOOTLOADER))"
```

After having programmed the node with the Blink application, we can try to verify the successful installation of the boot loader.

Once installed, we reset our node by turning it off then on. We should notice TOSBoot's execution by displaying a count-down sequence on the LEDs, with the red LED turning off last. Once all the LEDs have turned off, the Blink application should start blinking the red LED. The successful completion of this step is crucial for the reprogramming to work.

A.2 Installing DelugeDiff

To format the flash storage, we proceed as described in the tutorial. Next, we compile and install the `DelugeDiffBasic` application. We add the DelugeDiff library to the makefile like this:

```
PFLAGS += -I/opt/tinyos-1.x/tos/lib/DelugeDiff
```

The install process will install both the basic DelugeDiff application and TOSBoot. We should make sure to set the node ID appropriately when installing the application. For the feedback to work properly, every node should be assigned a unique ID. For example:

```
% make tinynode bsl.<x> install.9 TINYOS_NP=BNP
```

will set the node ID to 9 when installing the application. DelugeDiff will save the node ID so that it remains persistent across reboots between different program images.

A.3 Reprogramming with a New Program Image

We continue with the tutorial by pinging the node and installing `DelugeDiffBasic` as the golden image. To prepare the `Blink` code for network reprogramming, we add the `DelugeDiffC` component and wire it to `Main.StdControl`. Now we install this application in the network as in the tutorial. We choose image number 2 for the application. The output for 10 nodes should look something like this:

```
$ java net.tinyos.tools.DelugeD -i -ti=build/tinynode/
    tos_image.xml -in=2 -f -n=10
Pinging node ...
Connected to Deluge node.
Getting data for image [3] -----------------------
Ihex read complete:
  Total bytes = 34280
  Sections = 2
--------------------------------------------------
Replace empty image with:
  Image: 2
    Prog Name:   Blink
    Compiled On: Thu Nov 23 15:04:26 CET 2006
    Platform:    tinynode
    User ID:     andreas
```

```
    Hostname:    eth-0c224a10dfd
    User Hash:   0x79eb35f1
Injecting page [32] of [32] ...

Node 102: Image 2 updated
Node 108: Image 2 updated
Node 106: Image 2 updated
Node 104: Image 2 updated
Node 100: Image 2 updated
Node 103: Image 2 updated
Node 105: Image 2 updated
Node 109: Image 2 updated
Node 107: Image 2 updated
Node 101: Image 2 updated
update done!

--------------------------------------------------
DONE!
```

A.4 Updating with a New Program Image

In this section, we will update our network with a new application. The `Blink` application has already been installed on the node in image slot number 2. We choose the `CntToLeds` application for the reprogramming. We prepare it as described above by adding the `DelugeDiffC` component and compile it like this:

```
% make tinynode TINYOS_NP=BNP
```

Now, we have to create the delta file for this update process. This has to be done manually under Windows. We start by extracting the binary code:

```
% java net.tinyos.tools.DelugeD -w -ti=build/tinynode/
    tos_image.xml -o=CntToLedsDelugeDiff.bin
```

The binary code of the application is output in `CntToLedsDelugeDiff.bin`, specified with the `-o` parameter. We repeat the same procedure for our Blink application, resulting in the `BlinkDelugeDiff.bin` file. To generate the delta, we call the Xdelta compressor:

```
% xdelta3 -s BlinkDelugeDiff.bin CntToLedsDelugeDiff.bin
    Blink2CntToLeds.delta
```

We pass the binary source and target files to the encoder and get the final delta file, `Blink2CntToLeds.delta`.

To start the update process, we change to the `CntToLedsDelugeDiff` folder and call the java tool chain:

```
% java net.tinyos.tools.DelugeD -u -ti=build/tinynode/
    tos_image.xml -de=Blink2CntToLeds.delta
    -in=<image_number> -n=<number_of_nodes>
```

We start the update with the `-u` parameter, indicate the target image file with `-ti` and provide the delta file to use with the `-de` parameter. To pass the image number where we want the update installed we use the `-in` parameter and the number of nodes in the network is specified with `-n`.

Under Linux it works much easier, `-si` refers to the original image, the source image.

```
% java net.tinyos.tools.Deluge -u -si=<source_tos_image.xml>
    -ti=<target_tos_image.xml> -in=<image_number>
    -n=<number_of_nodes>
```

If we do the reprogramming with 10 nodes, we receive a feedback from each node when it has completely received the delta and when it has finished the update process. Again for ten nodes, the output could look as follows:

```
$ java net.tinyos.tools.DelugeD -u -ti=build/tinynode/
    tos_image.xml -de=Blink2CntToLeds.delta -in=2 -f -n=10
Pinging node ...
Connected to Deluge node.
Getting data for image [3] -----------------------
Ihex read complete:
  Total bytes = 34456
  Sections = 2
--------------------------------------------------
Update image:
  Image: 2
    Prog Name:    Blink
    Compiled On: Thu Nov 23 15:04:26 CET 2006
    Platform:     tinynode
    User ID:      andreas
    Hostname:     eth-0c224a10dfd
    User Hash:    0x79eb35f1
With image:
  Image: 2
    Prog Name:    CntToLeds
    Compiled On: Thu Nov 23 15:05:18 CET 2006
    Platform:     tinynode
    User ID:      andreas
    Hostname:     eth-0c224a10dfd
    User Hash:    0x79eb35f1
Injecting page [7] of [7] ...

Node 102: Update received
Node 103: Update received
Node 102: Image 2 updated
Node 106: Update received
Node 108: Update received
Node 104: Update received
```

```
Node 109: Update received
Node 101: Update received
Node 100: Update received
Node 108: Image 3 updated
Node 107: Update received
Node 103: Image 2 updated
Node 105: Update received
Node 106: Image 2 updated
Node 104: Image 2 updated
Node 109: Image 2 updated
Node 101: Image 2 updated
Node 107: Image 2 updated
Node 100: Image 2 updated
Node 105: Image 2 updated
update done!

--------------------------------------------------
DONE!
```

We can now resume and execute the reboot command as follows:

```
% java net.tinyos.tools.DelugeD -r -in=2
```

We specify the image slot number of the application we would like to program the node with by setting `-in` to 2. After a few moments, the node will begin counting quickly through the LEDs, signaling the programming process. Once complete, the node displays the count-down sequence and executes `CntToLedsDelugeDiff`. This application simply flashes the LEDs counting from 1 to 7. We can check it is running application by pinging the node again.

We have just successfully reprogrammed a network over the air. To demonstrate the usefulness of the Golden Image slot, we reset our node repeatedly in succession. After repeated resets, TOSBoot will flash all three LEDs simultaneously and reprogram your node. Connect this node to your computer and ping it using the Deluge Java tool. You should see that its executing image is now `DelugeDiffBasic` again.

A.5 Frequently Asked Questions

This section lists a set of common mistakes that users should avoid:

1. **I injected the reboot command, my node reboots, but does not reprogram itself to the new image.**

   Look carefully at the LEDs displayed by TOSBoot. If TOSBoot blinks the red LED three times before counting down, the system voltage is too low to safely reprogram the node. Replace your batteries and try again.
   If no LEDs blink at all after the reboot command has been injected, maybe TOSBoot is not or incorrectly installed. Test this as described in A.1. In case this fails, something is wrong with your boot loader.

2. **If I install DelugeDiff on a node where it has been installed on previously, the old images are still present on the node.**

   This happens sometimes because DelugeDiff loads the meta data stored in flash memory. You should always be aware of the fact that even the formatting application does not totally erase or rewrite flash memory.

3. `TOS_LOCAL_ADDRESS` **and** `TOS_GROUP_ID` **are not restored appropriately.**

   Important node state including `TOS_LOCAL_ADDRESS` and `TOS_GROUP_ID` are restored when `NetProg.init()` is called. It is important to remember that `NetProg.init()` should be called before using any of these values. To ensure this behavior, wire `DelugeC.StdControl` or `NetProgC.StdControl` to `Main.StdControl` and never reference `TOS_LOCAL_ADDRESS` and `TOS_GROUP_ID` in `StdControl.init()` of any module.

Bibliography

[ABF+00] M. Ajtai, R. Burns, R. Fagin, D. Long, and L. Stockmeyer. Compactly encoding unstructured input with differential compression, 2000.

[Aru04] M. (Umamaheswaran) Arumugam. Infuse: a tdma based reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 281–282, New York, NY, USA, 2004. ACM Press.

[Ber05] U.C. Berkeley. Deluge 2.0 - tinyos network programming manual, 2005. Retrieved: November 16, 2006 from U.C. Berkeley: `http://www.cs.berkeley.edu/~jwhui/research/deluge/deluge-manual.pdf`.

[BHS03] A. Boulis, Ch.-Ch. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 187–200, New York, NY, USA, 2003. ACM Press.

[Cad06] O. Caduff. Controlling wireless sensor networks. Master's thesis, Department of Computer Science, ETH Zurich, 2006.

[CT03] Inc. Crossbow Technology. Mote in-network programming user reference, 2003. Retrieved: November 16, 2006 from Crossbow Technology, Inc.: `http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf`.

[DFFMM06] H. Dubois-Ferrière, L. Fabre, R. Meier, and P. Metrailler. Tinynode: a comprehensive platform for wireless sensor network applications. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 358–365, New York, NY, USA, 2006. ACM Press.

[FTD06] G. Fuchs, S. Truchat, and F. Dressler. Distributed software management in sensor networks using profiling techniques. In *1st IEEE/ACM International Conference on Communication System Software and Middleware (IEEE COMSWARE 2006): 1st International Workshop on Software for Sensor Networks (SensorWare 2006)*, pages 1–6, jan 2006.

[Gai] J. Gailly. zlib compression library. Available at: `http://www.gzip.org/zlib/`.

[HC04] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[HM76] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report CSTR 41, Bell Laboratories, Murray Hill, NJ, 1976.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[HVT96] J. J. Hunt, K.-Ph. Vo, and W. F. Tichy. An empirical study of delta algorithms. In Ian Sommerville, editor, *Software configuration management: ICSE 96 SCM-6 Workshop*, pages 49–66. Springer, 1996.

[Jeo05] J. Jeong. Incremental network programming for wireless sensors. Master's thesis, EECS Department, University of California, Berkeley, November 21 2005.

[KMMV02] D. Korn, J. MacDonald, J. Mogul, and K. Vo. The VCDIFF Generic Differencing and Compression Data Format. RFC 3284 (Proposed Standard), June 2002.

[KP05] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks, 2005.

[LC02] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002. To appear.

[LPCS04] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks, 2004.

[Mac] J. MacDonald. xdelta compression tool. Available at: `http://www.xdelta.org/`.

[Mac00] J. MacDonald. File system support for delta compression, 2000.

[MGL+06] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 212–227, February 2006.

[NASZ05] V. Naik, A. Arora, P. Sinha, and H. Zhang. Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 277–286, Washington, DC, USA, 2005. IEEE Computer Society.

[Phi05] L. A. Phillips. Aqueduct: Robust and efficient code propagation in heterogeneous wireless sensor networks. Master's thesis, University of Colorado at Boulder, 2005.

[RL03] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM Press.

[SA05] Shockfish SA. Tinynode 584 / standard extension board user's manual, rev 1.1, 2005. Retrieved: November 16, 2006 from Shockfish SA: `http://www.tinynode.com/uploads/media/TinyNode/Users_Manual_rev11.pdf`.

[SHE03] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks, 2003.

[SM02] T. Suel and N. Memon. Algorithms for delta compression and remote file synchronization, 2002.

[Tic84] W. F. Tichy. The string-to-string correction problem with block moves. *ACM Trans. Comput. Syst.*, 2(4):309–321, 1984.

[TMS] D. Trendafilov, N. Memon, and T. Suel. zdelta: An efficient delta compression tool.

[TNCS02] Y.-Ch. Tseng, S.-Y. Ni, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. *Wirel. Netw.*, 8(2/3):153–167, 2002.

[Tri99] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[Wan04] L. Wang. Mnp: multihop network reprogramming service for sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 285–286, New York, NY, USA, 2004. ACM Press.

[Web05] TinyOS Website. Significant changes in tinyos between v1.1.13 and 1.1.14, 2005. Retrieved: November 16, 2006 from TinyOS Website: `http://www.tinyos.net/tinyos-1.x/doc/changes-minor-releases.html#1.1.14`.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[WK06] L. Wang and S. S. Kulkarni. Gappa: Gossip based multi-channel reprogramming for sensor networks. Technical Report MSU-CSE-06-8, Department of Computer Science, Michigan State University, East Lansing, Michigan, February 2006.

[WLLP01] B. Warneke, M. Last, B. Liebowitz, and K. S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.