Decoding Code on a Sensor Node

Pascal von Rickenbach and Roger Wattenhofer

Computer Engineering and Networks Laboratory, ETH Zurich, Switzerland
{pascalv, wattenhofer}@tik.ee.ethz.ch

Abstract. Wireless sensor networks come of age and start moving out of the laboratory into the field. As the number of deployments is increasing the need for an efficient and reliable code update mechanism becomes pressing. Reasons for updates are manifold ranging from fixing software bugs to retasking the whole sensor network. The scale of deployments and the potential physical inaccessibility of individual nodes asks for a wireless software management scheme. In this paper we present an efficient code update strategy which utilizes the knowledge of former program versions to distribute mere incremental changes. Using a small set of instructions, a delta of minimal size is generated. This delta is then disseminated throughout the network allowing nodes to rebuild the new application based on their currently running code. The asymmetry of computational power available during the process of encoding (PC) and decoding (sensor node) necessitates a careful balancing of the decoder complexity to respect the limitations of today's sensor network hardware. We provide a seamless integration of our work into Deluge, the standard TinyOS code dissemination protocol. The efficiency of our approach is evaluated by means of testbed experiments showing a significant reduction in message complexity and thus faster updates.

1 Introduction

Recent advances in wireless networking and microelectronics have led to the vision of sensor networks consisting of hundreds or even thousands of cheap wireless nodes covering a wide range of application domains. When performing the shift from purely theoretical investigations to physical deployments the need for additional network management services arises. Among other facilities this includes the ability to reprogram the sensor network [1,2]. Software updates are necessary for a variety of reasons. Iterative code updates on a real-world testbed during application development is critical to fix software bugs or for parameter tuning. Once a network is deployed the application may need to be reconfigured or even replaced in order to adapt to changing demands.

Once deployed, sensor nodes are expected to operate for an extended period of time. Direct intervention at individual nodes to install new software is at best cumbersome but may even be impossible if they are deployed in remote or hostile environments. Thus, network reprogramming must be realized by exploiting the network's own ability to disseminate information via wireless communication. Program code injected at a base station is required to be delivered to all nodes in its entirety. Intermediate nodes thereby act as relays to spread the software within the network. Given the comparatively small bandwidth of the wireless channel and the considerable amount of data to be distributed, classical flooding is prone to result in serious redundancy, contention, and collisions [3]. These problems prolong the update completion time, i.e. the time until all nodes in the network fully received the new software. Even worse, sensor nodes waste parts of their already tight energy budgets on superfluous communication.

Current code distribution protocols for sensor networks try to mitigate the broadcast storm problem by incorporating transmission suppression mechanisms or clever sender selection [4–6].

The radio subsystem is one of the major cost drivers in terms of energy consumption on current hardware platforms. Therefore, communication should be limited to a minimum during reprogramming in order not to reduce the lifetime of the network too much. Orthogonal to the above mentioned efforts the amount of data that is actually disseminated throughout the network should be minimized. Data compression seems to be an adequate answer to this problem. As knowledge about the application currently executed in the sensor network is present[1] differential compression, also known as delta compression, can be applied. Delta algorithms compress data by encoding one file in terms of another; in our case encoding the new application in terms of the one currently running on the nodes. Consequently, only the resulting delta file has to be transferred to the nodes which are then able to reconstruct the new application by means of their current version and the received delta. There exists a rich literature proposing a plethora of different algorithms for delta compression, e.g. [7–12]. These algorithms shine on very large files. However, neither time nor space complexity is crucial considering the small code size of today's sensor network applications. It is much more important to account for the asymmetry of disposable computational power at the encoder and the decoder. While almost unlimited resources are available to generate the delta file on the host machine special care must be taken to meet the stringent hardware requirements when decoding on the nodes.

In this paper we present an efficient code update mechanism for sensor networks based on differential compression. The delta algorithm is pursuing a greedy strategy resulting in minimal delta file sizes. The algorithm operates on binary data without any prior knowledge of the program code structure. This guarantees a generic solution independent of the applied hardware platform. We refrain from compressing the delta any further as this would exceed the resources available at the decoder. Furthermore, in contrast to other existing work we directly read from program memory to rebuild new code images instead of accessing flash memory which is slow and costly. The delta file is also structured to allow sequential access to persistent storage. All this leads to a lean decoder that allows fast and efficient program reconstruction at the sensor nodes.

Our work is tightly integrated into Deluge [5], the standard code dissemination protocol for the TinyOS platform. Deluge has proven to reliably propagate large objects in multi-hop sensor networks. Furthermore, it offers the possibility to store multiple program images and switch between them without continuous download. We support code updates for all program images even if they are not currently executed. Performance evaluations show that update size reductions in the range of 30% for major upgrades to 99% for small changes are achieved. This translates to a reprogramming speedup by a factor of about 1.4 and 100, respectively.

The remainder of the paper is organized as follows: After discussing related work in the next section, we give an overview of the code update mechanism in Section 3.

[1] This assumption is based on the fact that sensor networks are normally operated by a central authority.

Section 4 describes the update creation process as well as the decoder. In the subsequent section we present an experimental evaluation of our reprogramming service. Section 6 concludes the paper.

2 Related Work

The earliest reprogramming systems in the domain of wireless sensor networks, e.g. XNP [13], did not spread the code within the network but required the nodes to be in transmission range of the base station in order to get the update. This drawback was eliminated by the appearance of MOAP [4] which provides a multi-hop code dissemination protocol. It uses a publish-subscribe based mechanism to prevent saturation of the wireless channel and a sliding window protocol to keep track of missing information.

Deluge [5] and MNP [6] share many ideas as they propagate program code in an epidemic fashion while regulating excess traffic. Both divide a code image into equally sized pages, pipelining the transfer of pages and thus making use of spatial multiplexing. A bit vector is used to detect packet loss within a page. Data is transmitted using an advertise-request-data handshake. Deluge uses techniques such as a sender suppression mechanism borrowed from SRM [14] to be scalable even in high-density networks. In contrast, MNP aims at choosing senders that cover the maximum number of nodes requesting data. There have been various proposals based on the above mentioned protocols, e.g. [15, 16], that try to speed up program dissemination. However, all these approaches share the fact that the application image is transmitted in its entirety. This potentially induces a large amount of overhead in terms of sent messages but also in terms of incurred latency.

There have been efforts to update applications using software patches outside the sensor network community in the context of differential compression that arose as part of the string-to-string correlation problem [17]. Delta compression is concerned with compressing one data set, referred to as the *target image*, in terms of another one, called the *source image*, by computing a *delta*. The main idea is to represent the target image as a combination of copies from the source image and the part of the target image that is already compressed. Sections that cannot be reconstructed by copying are simply added to the delta file. Examples of such delta encoders include *vdelta* [7], *xdelta* [9], and *zdelta* [10]. They incorporate sophisticated heuristics to narrow down the solution space at the prize of decreased memory and time complexity as it is important to perform well on very large input files. However, these heuristics result in suboptimal compression. The *zdelta* algorithm further encodes the delta file using Huffman coding. This raises the decoder complexity to a level which does not match the constraints of current sensor network hardware. In [18], the *xdelta* algorithm is used to demonstrate the efficiency of incremental linking in the domain of sensor networks. However, the authors do not give a fully functional network reprogramming implementation but use the freely available *xdelta* encoder to evaluate the fitness of their solution. There exists other work in the domain of compilers and linkers trying to generate and layout the code such that the new image is as similar as possible to a previous image. Update-conscious compilation is addressed in [19] where careful register and data allocation strategies lead to significantly smaller difference files. In [20] incremental linkers are presented that optimize the object code layout to minimize image differences. All these

approaches are orthogonal to our work and can be integrated to further increase the overall system performance. We refrain from including one of them since they are processor specific and thus do not allow a generic solution.

Similar to the above mentioned delta algorithms, *bsdiff* [11] does also encode the target image by means of copy and insert operations. The algorithm does not search for perfect matches to copy from but rather generates approximate matches where more than a given percentage of bytes are identical. The differences inside a match are then corrected using small insert instructions. The idea is that these matches roughly correspond to sections of unmodified source code and the small discrepancies are caused by address shifts and different register allocation. Delta files produced by *bsdiff* can be larger than the target image but are highly compressible. Therefore, a secondary compression algorithm is used (in the current version *bzip2*) which makes the algorithm hardly applicable for sensor networks. The authors of [21] propose an approach similar to *bsdiff*. Their algorithm also produces non-perfect matches which are corrected using repair and patch operations. The patch operations work at the instruction level[2] to recognize opcodes having addresses as arguments which must be moved by an offset given by the patch operation. The algorithm shows promising results but depends on the instruction set of a specific processor.

In [8] *rsync* is presented that efficiently synchronizes binary files in a network with low-bandwidth communication links. It addresses the problem by using two-stage fingerprint comparison of fixed blocks based on hashing. An adaptation to *rsync* in the realm of sensor networks is shown in [22]. As both the source image and the target image reside on the same machine various improvements were introduced. The protocol was integrated into XNP. Besides the fact that XNP does only allow single-hop updates, the protocol does not overcome the limitations of *rsync* and performs well only if the differences in the input files are small.

FlexCup [23] exploits the component-based programming abstraction of TinyOS to shift from a monolithic to a modular execution environment. Instead of building one single application image FlexCup produces separate object files which are then linked to an executable on the sensor node itself. Thus, code changes are handled at the granularity of TinyOS components. This solution does no longer allow global optimizations. Furthermore, since the linking process requires all memory available at the sensor node, FlexCup is not able to run in parallel to the actual application. In [24], dynamic runtime linking for the Contiki operating system [25] is presented.

Besides TinyOS, there exist other operating systems for sensor networks which are inherently designed to provide a modular environment [1, 25]. They provide support for dynamic loading of applications and system services as a core functionality of the system. However, this flexibility implies additional levels of indirection for function calls which add considerable runtime overhead. The update process for changed components is limited to these components as they are relocatable and address independent.

Virtual machine architectures for sensor networks [26–28] push the level of indirection one step further. They conceal the underlying hardware to offer high-level operations to applications through an instruction interpreter. Updates are no longer native code but normally considerable smaller application scripts. This renders repro-

[2] Their work is based on the MSP430 instruction set.

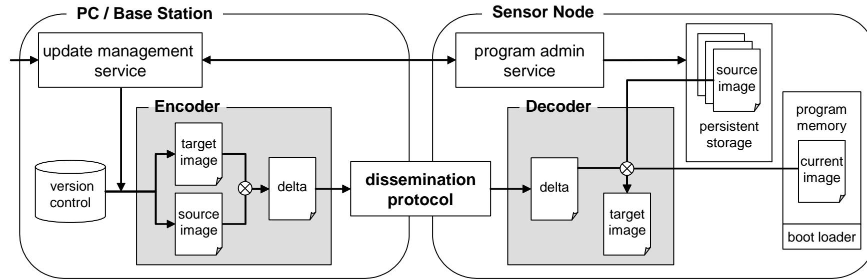

Fig. 1. Components involved in the process of wireless reprogramming.

gramming highly efficient. However, the execution overhead of a virtual machine is considerable and outweighs this advantage for long-running applications [24, 26].

A temporary alternative to supply in-network programmability inside the sensor network itself is to provide a parallel maintenance network [29]. This is particularly useful during the development process as one does not have to rely on the network being operational to update it. Furthermore, new protocols can be tested and evaluated without the reprogramming service distorting the results.

3 Overview

Updating code in wireless sensor network is a non-trivial task and requires the interaction of multiple system components. In general, application reprogramming can be broken down into three steps: image encoding, image distribution, and image decoding. Figure 1 shows a schematic view of all involved components and how they are interrelated in our code update mechanism. On the left-hand side, all services are consolidated that run on the host machine or base station, respectively. On the right, the required components on a sensor node are depicted. The dissemination protocol is responsible to reliably distribute the encoded update in the entire sensor network. We make use of Deluge as it is widely accepted as the standard dissemination protocol and has shown its robustness in various real-world deployments. We give a brief overview of Deluge's data management as it has direct implications on all other system components.[3]

Deluge enables a sensor node to store multiple application images. It divides a junk of the external flash memory (EEPROM) into slots, each of them large enough to hold one image. In conjunction with a bootloader Deluge is then able switch between these images. To manage the program image upload, Deluge divides images into pages of fixed size.[4] So far, Deluge transmits images at the page granularity. That is, all packets of the last page in use were distributed no matter how many of them actually containing data of the new application image. The residual space of the last page is thereby filled with zero bytes. This overhead of up to one kilobyte might be of minor concern if the application image is transmitted in its entirety. However, it becomes unacceptable in the context of small changes leading to delta files of only a few bytes.

[3] The interested reader is referred to [5] for a detailed description of Deluge.

[4] In the current version of Deluge one page sums up to 1104 bytes. In turn, this results in 48 data packets per page.

Deluge was therefore adapted to just transmit packets containing vital information about the new image. The remaining bytes of the last page are then padded with zeros on the sensor node itself to enable a 16-bit cyclic redundancy check on the pages. By requiring a node to dedicate itself to receiving a single page at a time, it is able to keep track of missing packets using a fixed-size bit vector. Packets also include CRC checksums. Redundant data integrity checks at both packet and page level is critical as erroneous data is otherwise propagated throughout the whole network due to the epidemic nature of Deluge.

The protocol also incorporates an administration service that allows the base station to retrieve information about all stored images including which one is currently running. On the host machine, Deluge offers an update management service to inject new images into the network. To allow differential updates a version control system is required at the host machine in order to know all application images currently residing on the sensor nodes. In the current version a file-system-based image repository is used to archive the latest program versions stored in each slot on the sensor nodes. If a new target image is supposed to be injected to a given slot, the update manager first queries the nodes to retrieve metadata about all loaded images. Based on this information a crosscheck in the version control system is performed to ensure that the latest image version for the requested slot is present in the repository. Once the validity of the source image in the repository is verified it is used as input for the delta encoder along with the target image. The encoder processes both images and generates the corresponding delta file. The delta is then disseminated using Deluge as if it was a normal application image. However, it is not stored in the designated slot of the target image but in an additional EEPROM slot reserved for delta files. Upon complete delta reception, a node starts the decoding process using additional information from external flash memory and program memory. The target image is thereby directly reconstructed in its intended EEPROM slot. In the meantime, the delta is further disseminated within the network. We now give a detailed description of the encoding algorithm employed on the host machine as well as of the decoder that resides on the sensor nodes.

4 Update Mechanism

All delta algorithms introduced in Section 2 use some kind of heuristic to speed up the generation of copy commands and consequently to reduce the overall execution time of the encoder. In [30] a greedy algorithm is presented that optimally solves the string-to-string correction problem which lies at the heart of differential updating. While its time complexity is undesirable for very large input files it poses no problem in the context of sensor networks where program size is limited to a few hundred kilobytes.[5] Hence, the design of our delta encoder is based on the findings in [30]. Before we give a detailed description of the encoder itself we specify the employed instruction set and how instructions are arranged in the delta file.

[5] The maximal application memory footprint of state-of-the-art sensor network hardware is limited to 48kB for nodes equipped with MSP430 microcontrollers or 128kB for ATmega128 platforms, respectively.

Instruction	Code	Arguments	Cost [bytes]
`shift`	xxxxx100	none	1
`run`	xxxxx101	byte to be repeated	2
`copy`	xxxxx110	start address	3
`add`	xxxxx111	data to be added	1+#bytes

Table 1. Instruction codes, arguments, and overall costs in bytes if the instructions reconstruct less than 32 bytes. The length of the instruction is encoded in the first five bits of the instruction code.

4.1 Delta Instructions and Delta File Organization

We adopt the set of delta instructions specified in VCDIFF [31] which is a portable data format for encoding differential data. It is proposed to decouple encoder and decoder implementations to enable interoperability between different protocol implementations. It distinguishes three types of instructions: `add`, `copy` and `run`. The first two instructions are straightforward; `add` appends a number of given bytes to the target image and `copy` points to a section in the source image to be copied to the target image. The `run` instruction is used to encode consecutive occurrences of the same byte efficiently. It has two arguments, the value of the byte and the number of times it is repeated. Making use of the fact that the target image is decoded into the same slot in external memory where the source image already resides, we introduce a fourth instruction. The `shift` instruction is used to encode sections of the image that have not changed at all from one version to the next. It is used to prevent unnecessary EEPROM writes. The only effect of a `shift` instruction is the adjustment of the target image pointer at the decoder.

These instructions are designed to minimize the overhead of the delta file. Each instruction code has a size of one to three bytes dependent on the number of bytes in the target image the corresponding instruction encodes. Table 1 comprises the arguments and costs of all four instruction types if they reconstruct less than 32 bytes of the target image. The actual length is directly encoded in the first 5 bits of the instruction code in this case. The cost of an instruction increases by one if the encoded fragment spans up to 255 bytes, or by two if it is larger than that, as the instruction length occupies one or two additional bytes, respectively.

We refrained from using the delta file organization as proposed in VCDIFF. It splits the file in three sections, one for data to be added, one for the actual instructions, and one for the addresses of the `copy` instructions. This enables better secondary compression of the delta file. As we try to keep the decoder complexity to a minimum to meet the nodes' hardware limitations no such compression is applied. We could still use the VCDIFF format without secondary compression. However, the fact that an instruction has to gather its arguments from different places within the delta file results in unfavorable EEPROM access patterns. Random access to external memory—as it would be the case if the VCDIFF format was employed—results in increased overhead during the decoding process. This is caused by the discrepancy between the small average delta instructions and the rather coarse-grained EEPROM organization. On average, the delta instruction length is below four bytes for all experiments described in Section 5. In contrast, external memory access is granted at a page granularity with

Operation	Current Draw	Time	Rel. Power Drain
Receive a packet	14 mA	5 ms	1
Send a packet	33 mA	5 ms	2.36
Read EEPROM page	4 mA	0.3 ms	0.017
Write EEPROM page	15 mA	20 ms	4.29

Table 2. Relative energy consumption of different operations in comparison to a packet reception for the TinyNode platform.

page sizes of 256 bytes for flash chips of modern sensor network hardware. As EEPROM writes are expensive (see Table 2) current flash storage incorporates a limited amount of cached memory pages to mitigate the impact of costly write operations.[6] However, it is important to notice that even though a read itself is cheap, it may force a dirty cache page to be written back to EEPROM which renders a read operation as expensive as a write.

To allow for the above mentioned EEPROM characteristics the delta file is organized by appending instructions in the order of their generation. That is, each instruction code is directly followed by its corresponding arguments. Furthermore, all instructions are ordered from left to right according to the sections they are encoding in the target file. This permits a continuous memory access during the execution of the delta instructions.

4.2 Delta Encoder

The severe hardware constraints of wireless sensor networks let our delta encoder differ in various points from common delta compression algorithms to optimize the decoding process. Besides the objective to minimize the delta file size one also has to consider the energy spent on reconstructing the new image at the nodes. In particular, special care has to be taken to optimize external flash memory access.

The only instruction that requires additional information from the source image to reconstruct its section of the target image is `copy`. To avoid alternating read and write requests between source image and delta file potentially causing the above discussed EEPROM cache thrashing problem we derive the data required by `copy` instructions directly from program memory. This decision has several implications. Most important, copies must be generated based on the currently executed image even if it is not identical to the image we would like to update. That is, the decoder is actually using a third input file, namely the currently executed image, to reconstruct the target image. Second, decoding is sped up since reading from program memory is fast in comparison to accessing external flash memory. Third, we are able to directly overwrite the source image in external memory without wasting an additional slot during the reconstruction process. This renders `shift` instructions possible. As a drawback, it is no longer allowed to use an already decoded section of the target image as origin for later copies. This potentially results in larger delta files. However, the aforementioned positive effects compensate this restriction.

[6] The Atmel AT45DB041B flash chip on the TinyNode platform has a cache size of two pages.

In a first phase, the encoder analyzes both source and target image. The algorithm runs simultaneous over both input files and generates `shift` instructions for each byte sequence that remains unchanged. Then, the target image is inspected and `run` instructions are produced for consecutive bytes with identical values. In a third pass, for each byte in the target image a search for the longest common subsequence in the source image is performed. A `copy` is then generated for each byte with a matching sequence of size at least three as `copy` instructions of length three or larger start to pay off compared to an `add`.

In a second phase a sweep line algorithm is employed to determine the optimal instruction set for the target image minimizing the size of the resulting delta. All instructions produced in the first phase reconstruct a certain section of the target image determined by their start and end address. The algorithm processes the image from left to right and greedily picks the leftmost instruction based on its start addresses. Then, the next instruction is recursively chosen according to the following rules. First, the instruction must either overlap with or be adjacent to the current instruction. Second, we choose the instruction among those fulfilling the previous requirement whose endpoint is farthest to the right. The instruction costs are used for tie breaking. To avoid redundancy in the delta file the new instruction is pruned to start right after the end of the current one if they overlap. If no instruction satisfies these demands an `add` is generated. These `add` instructions span the sections not covered by any of the other three instruction types. Once the algorithm reaches the end of the target image the delta file is generated according to the rules stated in the previous section.

4.3 Delta Decoder

The delta decoder is mapped as a simple state machine executing delta instruction in rotation. Prior to the actual decoding metadata is read from the head of the delta file. This information is appended by the encoder and contains the delta file length and additional metadata for the target image required by Deluge. The length is used to determine completion of the decoding. The metadata comprises version and slot information of the target image. This information is used to verify the applicability of the delta to the image in the given slot. In case of failure the decoding process is aborted and the image is updated traditionally without the help of differential reprogramming. If the delta is valid, the instructions are consecutively executed to rebuild the target image.

Once the decoder has fully reconstructed the image it signals Deluge to pause the advertisement process for the newly built image. This has the effect that the delta is disseminated faster within the network than the actual image enabling all nodes to reprogram themselves by means of the delta.

5 Experimental Evaluation

In this section we analyze the performance of our differential update mechanism on real sensor network hardware. The experiments were run on TinyNode 584 [32] sensor nodes operating TinyOS. The TinyNode platform comprises a MSP430 microcontroller

Setting	Target Size [bytes]	Delta Size[bytes]	Size Reduction	Encoding Time
Case 1	28684	322	98.88%	8453 ms
Case 2	27833	5543	80.08%	5812 ms
Case 3	28109	7383	73.73%	6797 ms
Case 4	34733	17618	49.28%	7563 ms
Case 5	21508	14904	30.70%	4781 ms

Table 3. Target and delta sizes for the different settings. Additionally, times required to encode the images are given.

featuring 10 kB of RAM and 48kB of program memory. Furthermore 512 kB external flash memory are available.

To prove the fitness of the proposed approach in a wide range of application scenarios five different test cases are consulted ranging from small code updates to complete application exchanges. Except one, all applications are part of the standard TinyOS distribution. Before evaluating the performance of our reprogramming approach the different test settings are discussed in the following.

Case 1: This case mimics micro updates as they occur during parameter tuning. We increase the rate at which the LED of the `Blink` application is toggled. This change of a constant has only local impact and should therefore result in a small delta.

Case 2: The `Blink` application is modified to facilitate concurrent program executions. The application logic is therefore encapsulated in an independent task (see `BlinkTask`). This leads to additional calls to the TinyOS scheduler and a deferred function invocation.

Case 3: The `CntToLeds` application, which shows a binary countdown on the LED's, is extended to simultaneously broadcast the displayed value over the radio (`CntToLedsAndRfm`). This is a typical example of a software upgrade that integrates additional functionality.

Case 4: In this setting `Blink` is replaced with the `Oscilloscope` application. The latter incorporates multi-hop routing to convey sensor readings towards a base station. Both application share a common set of system components. This scenario highlights the ability of our protocol to cope with major software changes.

Case 5: Here we switch from `Blink` to `Dozer` [33], an energy-efficient data gathering system. Among other features, Dozer employs a customized network stack such that the commonalities between the two applications are minimal. Furthermore, `Dozer` is the only application in this evaluation that has no built-in Deluge support.

The `Blink` application produces a memory footprint of 24.8 kB in program memory and 824 bytes RAM using the original Deluge. In comparison, the enhanced Deluge version including delta decoder sums up to 27.6 kB ROM and 958 bytes of RAM. Consequently, our modifications increase the memory footprint of an application by 2.8 kB in program memory and 134 bytes of RAM.

The performance of our delta encoder for all five cases is shown in Table 3. For Case 1, the encoder achieves a size reduction by a factor of 100. Actually, the mere delta

Setting	#`shift`	avg. size	#`run`	avg. size	#`copy`	avg. size	#`add`	avg. size
Case 1	5	57334	0	0	0	0	5	2.80
Case 2	16	79.19	3	244	884	27.43	859	1.83
Case 3	71	26.01	5	91	1183	20.13	1153	1.72
Case 4	30	37.13	12	40.32	2757	9.68	2472	2.64
Case 5	25	7.78	20	54.70	2219	6.44	1858	3.19

Table 4. The number of occurrences of each instruction type for all scenarios. Furthermore, the average number of bytes encoded by one instruction is given.

is only 32 bytes long. The other 290 bytes consist of metadata overhead introduced by Deluge such as 256 bytes of CRC checksums. As already mentioned in the case descriptions, the increasing delta sizes indicate that the similarity between source and target image decreases from Case 1 to 5. The delta file produced due to major software changes is still only about half the size of the original image. Moreover, even if we replace an application with one that has hardly anything in common with the former, such as in Case 5, the encoder achieves a size reduction of about 30%. Table 3 also contains the execution times of the encoder for the five different scenarios. Note that the encoding was computed on a customary personal computer. The encoding process for the considered settings takes up to nine seconds. Compared to the code distribution speedup achieved by smaller delta files this execution time is negligible.

Table 4 shows the number of occurrences of all four instruction types in all five settings. It also contains the average number of bytes covered by one instruction. One can see that the modifications in Case 1 are purely local as only 14 bytes have to be overwritten and the rest of the image stays untouched. For the other four scenarios, `copy` and `add` instructions constitute the dominating part of the delta files. It is interesting to see that the average size of a `copy` is larger if the source and target images have a higher similarity. The opposite is true for the `add` instruction. In the case of minor application updates many code blocks are only shifted to a different position within the code image but not changed at all. This fact is exploited by the `copy` instructions enabling a relocation of these sections with constant overhead. However, if the new application is completely unrelated to the one to be replaced as in Case 5, the image exhibits less opportunities for copies. Consequently, more `add` instructions are necessary to rebuild the target image.

To evaluate the decoder we measure the reconstruction time of the target image on a sensor node. Table 5 shows the decoding time for all cases dependent on the available buffer size at the decoder. If no input buffer is available, each delta instruction is read separately from external memory before it is processed. If an input buffer is allocated, the decoder consecutively loads data blocks of the delta file from EEPROM into this buffer. Before a new block of data is fetched, all instructions currently located in the buffer are executed. Similar to the input buffer handling, the decoder writes the result of a decode instruction directly to external memory if no output buffer is present. In contrast, if an output buffer is available, it is filled with the outcomes of the processed delta instructions and only written back to EEPROM if it is full. We

Setting	none | none	256 | none	256 | 256	128 | 128	64 | 64	32 | 32	16 | 16
Case 1	1.20 s	1.24 s	1.24 s	1.24 s	1.24 s	1.23 s	1.23 s
Case 2	7.03 s	5.05 s	4.11 s	4.20 s	4.38 s	4.74 s	5.40 s
Case 3	8.51 s	5.52 s	4.25 s	4.36 s	4.54 s	5.02 s	5.74 s
Case 4	15.14 s	8.68 s	5.63 s	5.79 s	6.06 s	6.82 s	7.84 s
Case 5	11.27 s	6.47 s	4.08 s	4.15 s	4.35 s	4.74 s	5.45 s

Table 5. Time to reconstruct the target image on a sensor node as a function of the available input and output buffer sizes in bytes at the decoder (input | output).

limit the maximum buffer size to 256 bytes thereby matching the EEPROM page size of the TinyNode platform.

For Case 1 the decoding process takes approximately 1.2 seconds no matter which buffer strategy is applied. This can be explained by the fact that only 10 delta instructions are involved (see Table 4) where five of them are `shift` instructions which do not lead to EEPROM writes. In contrast, the decoder takes about 15 seconds to update from `Blink` to the `Oscilloscope` application if neither input nor output buffers are used. However, decoding time decreases to 8.68 or 5.63 seconds if an input buffer of 256 bytes or both, input and output buffers of size 256 bytes are employed, respectively. That is, decoding with maximum input and output buffer reduces the execution time by a factor of 2.7 in Case 4.

Due to the promising results with large buffer sizes, we also study the impact of varying buffer sizes on the decoding speed. The reconstruction times for all scenarios were evaluated for buffer sizes of 16, 32, 64, 128, and 256 bytes, respectively. Table 5 shows that execution times increase with decreasing buffer sizes. However, the increases are moderate: Reducing the buffers from 256 to 16 bytes—and thus saving 480 bytes of RAM—results in an at most 40% longer decoding time. Furthermore, the execution times with buffers of 16 bytes are roughly the same as if only an input buffer of 256 bytes is used.

6 Conclusions

Sensor networks are envisioned to operate over a long period of time without the need of human interaction. Once deployed, remote reprogramming of the sensor network is therefore crucial to react to changing demands. This paper introduces an efficient code update strategy which is aware of the limitations and requirements of sensor network hardware. We exploit the fact that the currently running application on the sensor nodes is know at the time a new program is supposed to be installed. Differential compression is employed to minimize the amount of data that has to be propagated throughout the network. The proposed delta encoder achieves data size reductions from 30% to 99% for a representative collection of update scenarios. The delta decoder is integrated into the Deluge framework which guarantees reliable data dissemination within the sensor network. The energy spent for the image distribution is directly proportional to the transmitted amount of data. Thus, our system reduces the power consumption of the code dissemination by the same percentage as it compresses the input data.

References

1. C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A Dynamic Operating System for Sensor Nodes," in *Int. Conference on Mobile Systems, Applications, and Services (MobiSys), Seattle, Washington, USA*, 2005.
2. Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming Wireless Sensor Networks: Challenges and Approaches," *IEEE Network*, vol. 20, no. 3, pp. 48–55, 2006.
3. S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu, "The Broadcast Storm Problem in a Mobile Ad Hoc Network," in *Int. Conference on Mobile Computing and Networking (MobiCom), Seattle, Washington, USA*, 1999.
4. T. Stathopoulos, J. Heidemann, and D. Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks," UCLA, Tech. Rep. CENS-TR-30, 2003.
5. J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Int. Conference on Embedded Networked Sensor Systems (SENSYS), Baltimore, Maryland, USA*, 2004.
6. S. S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," in *Int. Conference on Distributed Computing Systems (ICDCS), Columbus, Ohio, USA*, 2005.
7. J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta Algorithms: An Empirical Analysis," *ACM Trans. Software Engineering and Methodology*, vol. 7, no. 2, pp. 192–214, 1998.
8. A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," Ph.D. dissertation, Astralian National University, 1999.
9. J. MacDonald, "File System Support for Delta Compression," Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.
10. D. Trendafilov, N. Memon, and T. Suel, "zdelta: An Efficient Delta Compression Tool," Polytechnic University, Tech. Rep. TR-CIS-2002-02, 2002.
11. C. Percival, "Naive Differences of Executable Code," 2003. [Online]. Available: http://www.daemonology.net/papers/bsdiff.pdf
12. R. C. Agarwal, K. Gupta, S. Jain, and S. Amalapurapu, "An approximation to the greedy algorithm for differential compression," *IBM J. of Research and Development*, vol. 50, no. 1, pp. 149–166, 2006.
13. C. T. Inc., "Mote In-Network Programming User Reference," 2003. [Online]. Available: http://www.xbow.com
14. S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang, "A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing," *IEEE/ACM Trans. Networking*, vol. 5, no. 6, pp. 784–803, 1997.
15. R. Simon, L. Huang, E. Farrugia, and S. Setia, "Using multiple communication channels for efficient data dissemination in wireless sensor networks," in *Int. Conference on Mobile Adhoc and Sensor Systems (MASS), Washington, DC, USA*, 2005.
16. R. K. Panta and I. K. S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," in *Int. Conference on Computer Communications (INFOCOM),Anchorage, Alaska, USA*, 2007.
17. R. A. Wagner and M. J. Fischer, "The String-to-String Correlation Problem," *J. ACM*, no. 21, pp. 168–173, 1974.
18. J. Koshy and R. Pandey, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," in *European Workshop on Wireless Sensor Networks (EWSN), Istanbul, Turkey*, 2005.
19. W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: Update-Conscious Compilation for Energy Efficiency in Wireless Sensor Networks," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA*, 2007.
20. C. von Platen and J. Eker, "Feedback Linking: Optimizing Object Code Layout for Updates," in *ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES),Ottawa, Ontario, Canada*, 2006.
21. N. Reijers and K. Langendoen, "Efficient Code Distribution in Wireless Sensor Networks," in *Int. Conference on Wireless Sensor Networks and Applications (WSNA), San Diego, California, USA*, 2003.

22. J. Jeong and D. Culler, "Incremental Network Programming for Wireless Sensors," in *Int. Conference on Sensor and Ad Hoc Communications and Networks (SECON), Santa Clara, California, USA*, 2004.
23. 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 *European Conference on Wireless Sensor Networks (EWSN), Zurich, Switzerland*, 2006.
24. A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks," in *Int. Conference on Embedded Networked Sensor Systems (SENSYS), Boulder, Colorado, USA*, 2006.
25. A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Int. Conference on Local Computer Networks (LCN), Orlando, Florida, USA*, 2004.
26. P. Levis and D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *ACM SIGOPS Operating System Review*, vol. 36, no. 5, pp. 85–95, 2002.
27. P. Levis, D. Gay, and D. Culler, "Active Sensor Networks," in *Int. Conference on Networked Systems Design and Implementation (NSDI), Boston, Massachusetts, USA*, 2005.
28. R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava, "Multi-Level Software Reconfiguration for Sensor Networks," in *Int. Conference on Embedded Software (EMSOFT), Seoul, Korea*, 2006.
29. M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum, "Deployment Support Network - A Toolkit for the Development of WSNs," in *European Conference on Wireless Sensor Networks (EWSN), Delft, Netherlands*, 2007.
30. C. Reichenberger, "Delta Storage for Arbitrary Non-Text Files," in *Int. Workshop on Software Configuration Management (SCM), Trondheim, Norway*, 1991.
31. D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The VCDIFF Generic Differencing and Compression Data Format," RFC 3284 (Proposed Standard), June 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3284.txt
32. H. Dubois-Ferrier and R. Meier and L. Fabre and P. Metrailler, "TinyNode: a comprehensive platform for wireless sensor network applications," in *Int. Conference on Information Processing in Sensor Networks (IPSN), Nashville, Tennessee, USA*, 2006.
33. N. Burri, P. von Rickenbach, and R. Wattenhofer, "Dozer: Ultra-Low Power Data Gathering in Sensor Networks," in *Int. Conference on Information Processing in Sensor Networks (IPSN), Cambridge, Massachusetts, USA*, April 2007.