Efficient reprogramming of wireless sensor networks using incremental updates

Milosh Stolikj, Pieter J. L. Cuijpers, and Johan J. Lukkien
Dept. of Mathematics and Computer Science, Eindhoven University of Technology,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

***Abstract*—Software reprogramming enables users to extend or correct functionality of a sensor network after deployment, preferably at a low cost. This paper investigates the improvement of energy efficiency and delay of reprogramming, at low resource cost. As enabling technologies data compression and incremental updates are used. Algorithms for both approaches are analyzed, as well as their combination, applied to resource-constrained devices. All algorithms are ported to the Contiki operating system, and profiled for different types of reprogramming. The presented results show that there is a clear trade-off between performance and resource requirements. Furthermore, the best reprogramming approach depends on the type of update. Experimentally, VCDIFF, or the combination of Lempel-Ziv-77/FastLZ for compression with BSDIFF for delta encoding, have been identified as the best possible options.**

I. Introduction

An important feature of networks of resource constrained devices is *reprogramming*, i.e. the capability to change software functionality at run time. Reprogramming is important both during development, for fast prototyping and debugging, and after deployment, for adapting functionality.

Software changes come in the form of updates, consisting of new applications, bug fixes, operating system updates or modified parameters. In remote reprogramming, updates are assembled at a host machine outside of the network. Then, the update has to be spread through the network, reaching every intended node. Due to the large size of the updates and high number of nodes in a network, most solutions for remote reprogramming suffer from long delays and high energy usage.

In this paper, we explore means for reducing the size of updates for remote reprogramming. Motivated by the fact that the processor consumes significantly less energy than the wireless radio, we investigate how much energy and time can be saved by reducing the updates as much as possible. Furthermore, the focus is on two approaches: 1) applying data compression algorithms directly to updates; and 2) using incremental updates, i.e. exchanging only the difference between two consecutive software versions, captured in scripts called *deltas*. Deltas have a highly compressible structure and they can improve the performance of data compression algorithms.

We selected five data compression algorithms along with three algorithms for incremental updates, and analyzed their applicability on resource constrained devices. The applicability is quantified through metrics which determine the resources required to host and execute the algorithms. Furthermore, we analyzed the performance of each data compression algorithm being used alone, or in combination with any of the three incremental update algorithms. Surprisingly, our results show that using only data compression can degrade performance, while incremental updates always give improvements.

Our contributions are threefold. First, we give clear measurements of how much resources are required to implement each algorithm. Second, we emphasised the trade-off between energy usage, delay and resource requirements in the selection process for compression algorithms intended for resource constrained devices. Finally, based on the results of the experiments, we selected the best possible combinations between data compression and incremental update, and populated a decision tree for selecting the best combination given resource and infrastructure requirements.

This work is supported in part by the Dutch P08 SenSafety Project, as part of the COMMIT program.

II. Related work

Optimizing software reprogramming has been extensively studied in wireless sensor networks. Modular operating systems are an improvement over non-modular systems by supporting dynamic linking and loading. This way, systems such as Contiki [1], allow partial executables to be deployed and executed at run time, without flashing the firmware. However, since the partial executables contain symbol and relocation tables, they can still be large in size for reliable transfer in lossy wireless networks.

Apart from complete firmware reprogramming [2] [3], alternative methods have been developed for updating non-modular systems. Virtual machines and middle-ware layers (Maté [4], OSAS [5]) overcome limitations of large updates for distribution by running interpreted code. Since byte code is much smaller compared to compiled binary code, updates in these systems can be easily distributed. The downside of this approach is that interpreted execution is slower and some resources are always used by the virtual machine. And still, the problem of large updates is present if the operating system or the virtual machine engine need to be updated.

Another approach to reprogramming is to use incremental updates of firmware images [6]. In [7], modified versions of the rsync and XNP protocols are used for generating deltas and their dissemination, respectively. Zephyr [8] adds application-level modifications to decrease the difference between consecutive application versions, then produces deltas with *rsync*. In [9], a tool similar to the *UNIX diff* is used to create

978-1-4244-9529-0/13/$31.00 ©2013 IEEE

deltas between versions. It extends the delta functionality with two new instructions, which enable more efficient coding of the differences. While these studies emphasise the benefits of using incremental updates, they use solely one algorithm, without evaluating whether a better option exists. In this paper, we connect incremental updates with data compression to give a broader view of options for optimizing the size of updates.

Data compression has been previously considered in sensor networks, mostly for data gathered from sensors [10] [11]. In [12], several algorithms are compared on desktop machines, for compressing data from two test beds. Similarly, in [13] compression algorithms are compared on ELF executables for the Contiki operating system. Since sensed data is less compressible than binary data, the reported results do not apply to software updates. Furthermore, during upgrades, only decompression is needed on resource-constrained devices. In this paper, we extend our previous evaluation of data compression algorithms for sensor networks [14] with multiple algorithms for incremental update. Here we emphasise the importance of the combination of both, and demonstrate which combinations are preferable in different scenarios for reprogramming.

III. Methodology

There are two common approaches for reducing the size of data in software updates: using data compression and incremental updates. Next, we discuss both approaches individually.

A. Performing updates using data compression

Compression, and accordingly decompression, is added to the update process as shown in Figure 1. It is an intermediate phase with the aim to encode information with fewer bits than the original representation. Data compression is done outside of the sensor network, so only decompression is needed on sensor nodes. Furthermore, since in executable data every bit is equally important, only lossless algorithms can be used.

While many data compression algorithms are available, most of them are inapplicable to sensor nodes due to high resource demands. Previously [14], we identified five Lempel-Ziv (LZ) variants as suitable for resource constrained devices. We used the same algorithms for the experiments in this work as well.

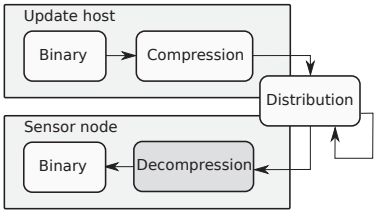


Fig. 1. Overview of the update process when using data compression.

B. Performing updates incrementally

Most changes in software come in the form of incremental updates, which either add additional functionality or modify values of existing parameters. The *old* and *new* version share most of the code base, and the difference between them is significantly smaller than the size of the application itself.

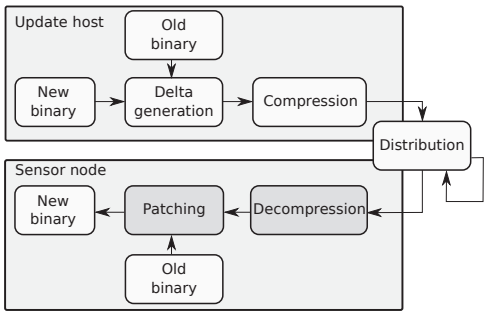


Fig. 2. Overview of the update process when using incremental updates.

Algorithms for delta encoding exploit this behaviour by extracting and distributing only the differences between both versions. The delta contains instructions and data, which are used to reconstruct the new version from the old one, a process called patching. Delta encoding algorithms differ in how the delta is constructed and how the differences are detected. Similar to data compression, the delta creation is done outside of the sensor network, and only patching functionality needs to be added on the sensor nodes. Next, the three most widely used delta encoding algorithms are analyzed.

Rsync, and the corresponding RDIFF algorithm [15], use non-overlapping fixed-sized blocks for matching identical data between the old and new version. Both versions are segmented into blocks, and for each one, a rolling-checksum and a MD5 checksum are computed. Based on these checksums, the delta is constructed of either references to blocks that already exist in the old version, or the entire content of new or changed blocks. While the rolling checksum is implemented to be as fast as possible, a MD5 checksum is not appropriate for sensor nodes. A weakness of the algorithm is that if two blocks differ in even one byte, the entire block has to be present in the delta.

VCDIFF [16] is a format for encoding the difference between two data sets. The original idea for it comes from the Lempel-Ziv 77 algorithm - the old and new version are concatenated; then the resulting stream is compressed using LZ77 or a similar algorithm. From the output, the first part, which corresponds to the old version, is omitted, leaving only the instructions for the decoder to decompress the new version. VCDIFF features a detailed byte-code instruction set, consisting of a small number of instructions, which can be used in different addressing modes, accessing both the old and the new data. In this paper, we use Xdelta [17] as an encoder for generating VCDIFF deltas. It reduces the delta size by optimizing the generated instruction set, removing completely covered instructions and merging small instructions into one.

BSDIFF [18] uses two passes to construct deltas. In the first pass, completely identical blocks are found in the two versions. Next, the exact matches are expanded in both directions, such that every prefix/suffix of the extension matches in at least half of its bytes. These matches roughly correspond to modified lines of code. The delta is then constructed of three parts: a control block of commands for reconstructing the new version; a diff block of bytewise differences between approximate matches and an extra block, consisting of new data. When

TABLE I
TEST SCENARIOS AND DATA SIZE OF FIRMWARE IMAGES AND ELF EXECUTABLES.

Test	Description	Type	Starting size	Final size
1	OS update (Contiki 2.3 → 2.4)	Firmware	22,924	20,624
2	OS update (Contiki 2.4 → 2.5)	Firmware	20,624	22,980
3	New application (OSAS 2.0)	Firmware	22,980	39,112
4	Application update (OSAS 1.0 → 2.0)	Firmware	37,796	39,112
5	Application update (OSAS 1.0 → 2.0)	ELF executable	25.784	26.712
6	Parameter change (OSAS 2.0 → 2.1)	Firmware	39.112	39.112
7	Parameter change (OSAS 2.0 → 2.1)	ELF executable	26,712	26,712

the old and the new version are very similar, the diff block consists of large series of zeroes, which are easily compressed.

All delta encoding algorithms use compression to reduce the delta's size. Therefore, by adding delta encoding, a sensor node is reprogrammed as in Figure 2. It can be seen as a pre-processor; an initial phase of data compression algorithms.

IV. EVALUATION

For algorithms running on resource-constrained devices, four metrics are relevant: code size of the algorithm, memory used during execution, energy and delay. The size of compressed data and execution time are two additional factors which directly determine energy usage and delay.

The reduction in size of the compressed data is quantified through the compression ratio. It is defined as the reduction in size relative to the uncompressed data:

$$compr\_ratio = (1 - \frac{compressed\_size}{uncompressed\_size}) * 100 \quad (1)$$

Consequently, higher values mean smaller compressed files, hence better performance.

Decompressing data requires a certain amount of processor cycles. A high number of processor cycles would result in large decompression times. As a result, regardless of processor speed, this value should be as low as possible. The importance of this metric is captured through the energy and delay models.

Memory is limited in resource-constrained devices. This includes both memory required for holding the code, which is stored in internal flash memory (ROM), and memory required during execution, in RAM. Algorithms running on sensor nodes must have a small code footprint, up to a couple of kilobytes, and use little memory during execution.

We estimate energy usage through a model which uses the amount of time spent during computation and transmission of data [19]. This is a lower bound of the real energy usage; we assume that forwarding is done immediately, without additional processing, and we ignore MAC protocol behavior. Adding those variables, will result in higher energy usage for transmission, penalising communication even further. We consider a star topology for update, where the central node receives the update and then sends it to h neighbours. We calculate energy usage of the central node as:

$$E = k_{\mathtt{err}} * \lceil \frac{data\_size}{payload\_size} \rceil * (E_{\mathtt{rx}} + h * E_{\mathtt{tx}}) + E_{\mathtt{cpu}}, \quad (2)$$

where $k_{\mathtt{err}}$ is the average number of times each packet is sent due to errors in the radio medium, $data\_size$ is the size of the data for transmission, $payload\_size$ is the maximum packet size, $E_{\mathtt{rx/tx}}$ is the energy required to receive/send one packet and E_{cpu} is the energy required for post-processing. Communication energy is expressed as $E_{\mathtt{rx/tx}} = t_{\mathtt{rx/tx}} * I_{rx/tx} * V$, where $t_{rx/tx}$ is the amount of time that the wireless radio is in listening/sending state. We simplify the model by assuming that during reception, the radio chip is turned on for the same amount of time as during sending, though it draws more current [20]. This corresponds to factory values of various radio chipsets, such as the CC2420. Similarly, processing energy is calculated as $E_{\mathtt{cpu}} = I_{\mathtt{cpu}} * V * t_{\mathtt{cpu}}$, where $t_{\mathtt{cpu}}$ is the amount of processing time.

We estimate the time needed for the central node to receive an update, apply it locally and distribute it to its neighbours, with a similar model to the one used for energy estimation. Again we estimate a lower bound of the delay, since we assume that forwarding is done immediately, and that the MAC protocol does not introduce additional overhead:

$$D = k_{\mathtt{err}} * \lceil \frac{data\_size}{payload\_size} \rceil * (t_{\mathtt{rx}} + h * t_{\mathtt{tx}}) + t_{\mathtt{cpu}}. \quad (3)$$

We consider three cases of energy usage and delay during reprogramming: 1) neither compression nor incremental updates is used ($t_{\mathtt{cpu}} = 0$); 2) only compression is used ($t_{\mathtt{cpu}} = t_{\mathtt{dcmp}}$) and 3) both compression and incremental update is used ($t_{\mathtt{cpu}} = t_{\mathtt{dcmp}} + t_{\mathtt{patch}}$).

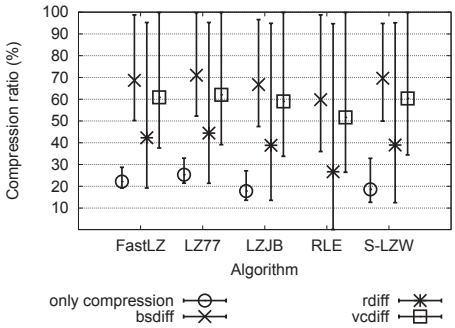
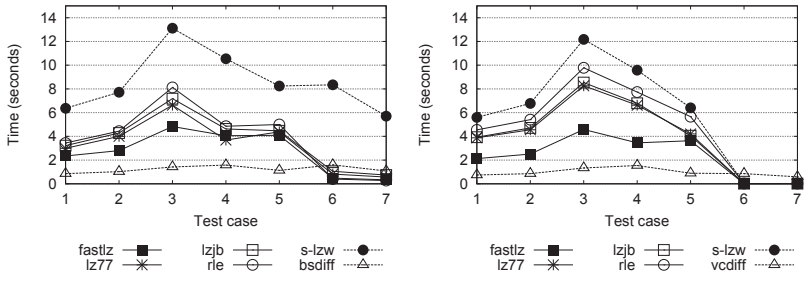


Fig. 3. Minimum, maximum and average measured compression ratio.

A. Experimental setup and workflow

In our experiments, we use the Contiki operating system, running on Crossbow TelosB nodes [21], with the Open Service Architecture for Sensors (OSAS) [5] application. The node contains an 8 MHz TI MSP430 microcontroller with the Chipcon CC2420 radio transceiver. It has 48 KB program flash memory, 10 KB of RAM and 1 MB external flash.



(a) BSDIFF (b) VCDIFF

Fig. 4. Time required for decompressing and applying a BSDIFF (a) and VCDIFF (b) patch.

We considered seven test scenarios for remote reprogramming, shown in Table I. For each test case, both the initial version and the new version are available. First, we compress the new version directly. Then, we produce a delta using each delta encoding algorithms, and apply compression to it. We measure the compression ratio of the compressed delta with respect to the size of the new version. Finally, we measure the remaining metrics for decompression and patching.

All algorithms were adapted to be run on the TelosB nodes[1]. Input and output data is stored on the external serial flash and is accessed through the Coffee file system [22]. All tests were executed 10 times, and timed using the Contiki clock module.

B. Results

Next we will discuss each of the aforementioned metrics.

1) Compression ratio: Compression ratio is a factor which gives a strong indication what to expect from a compression algorithm in terms of energy and delay savings. As illustrated on Figure 3, due to the diverse input samples, the compression ratio varies significantly between different test cases.

In general, the compression ratio metric implies that incremental updates make significant difference in the performance of compression algorithms. Depending on the approach and type of updates that need to be compressed, between 37% and 99% compression ratio can be achieved. Most compression algorithms behave similarly, with not more than 10% difference between them. The obvious exception was Run Length Encoding as the worst compressor.

Using BSDIFF showed higher compression ratio compared to the other delta encoding algorithms in all except the last two scenarios, in which VCDIFF produced smaller deltas. RDIFF was consistently inferior to the other two algorithms, and was therefore omitted from the subsequent experiments.

2) Memory requirements: This metric determines the memory resources required to add decompression and delta encoding support. It can be divided in two parts - memory required for holding the code, which is stored in internal flash memory (ROM), and memory required during execution, in RAM.

Table II shows code size and memory usage for the decompression and incremental update algorithms, ported to the Crossbow TelosB motes. The code size corresponds to the size of the .text segment of the ELF binary. Memory is the sum of static memory and maximum stack used during execution.

From the table, it is evident that Run Length Encoding, Lempel-Ziv 77 and LZJB are lightweight in terms of both code size and memory usage during execution; FastLZ has a larger code base, but still uses little stack space. Finally, Sensor-LZW has the largest code base and uses the most memory of all decompression algorithms.

The memory footprint of BSDIFF is small, both in code size and memory usage. On the other hand, VCDIFF has a significantly larger code base, along with large memory footprint, mostly for storing the instruction cache.

TABLE II
CODE AND MEMORY FOOTPRINT OF DIFFERENT ALGORITHMS, USING A TWO BYTE BUFFER

Algorithm	Code (bytes)	Memory (bytes)
fastlz	878	145
lz77	376	144
lzjb	424	140
rle	198	131
s-lzw	1.281	2502
bsdiff	560	158
vcdiff	2.261	1714

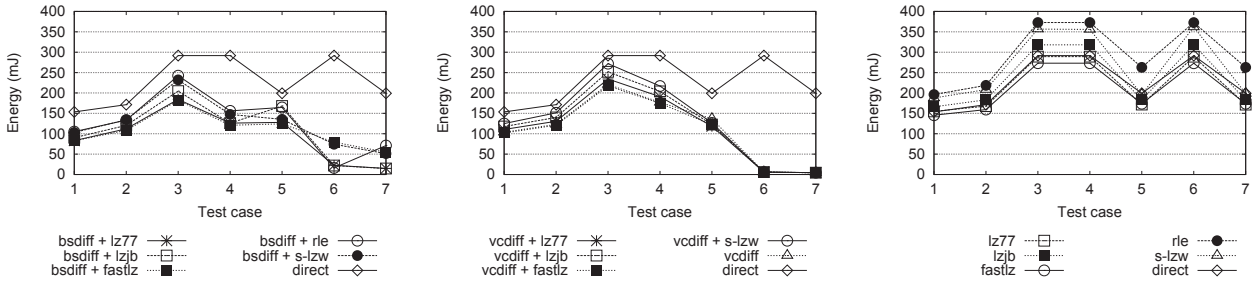
3) Processing requirements: The time required to decompress the BSDIFF/VCDIFF deltas is shown in Figure 4. In all cases, Sensor-LZW was the slowest algorithms. LZ77 and LZJB had similar execution times, while RLE had significantly worse performance while decompressing VCDIFF deltas. This comes down to the nature of the VCDIFF algorithm - run length encoding is done while the delta is generated. Finally, on average, FastLZ was the fastest algorithm.

BSDIFF and VCDIFF have comparable execution time when 128 byte buffers are used. VCDIFF is slightly faster in the last two test scenarios (parameter change), due to the smaller delta produced.

4) Energy estimation: For reprogramming one node, using only compressed updates (Figure 5c) is more energy efficient than sending data directly, only when FastLZ is used. The additional processing introduced by LZ77 in some cases pays off; all other algorithms require more energy.

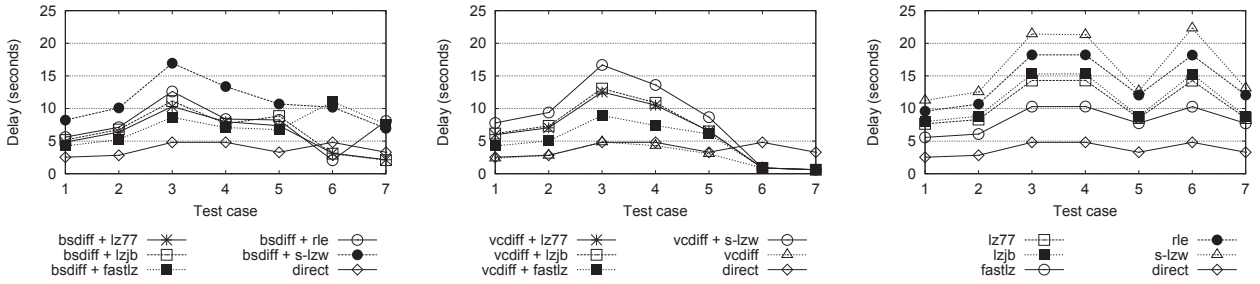
On the other hand, the combination of any compression algorithm with either BSDIFF (Figure 5a) or VCDIFF (Figure 5b) results in significant reductions in energy usage. For test cases 1 to 5, highest energy savings are achieved using

[1]The source code of the algorithms are available at http://www.win.tue.nl/~mstolikj/compression/. The port of VCDIFF to the MSP430 microcontroller was kindly provided by Nicolas Tsiftes.



(a) Decompression with BSDIFF patch (b) Decompression with VCDIFF patch (c) Only decompression

Fig. 5. Energy estimation using only decompression (c) and both patching and decompression (a, b). (Constants: $h = 1$, $k_{\mathrm{err}} = 1$, $payload\_size = 114$, *buffer size* = 128). "Direct" shows the energy usage of transmitting the data directly, without processing.



(a) Decompression with BSDIFF patch (b) Decompression with VCDIFF patch (c) Only decompression

Fig. 6. Delay estimation using only decompression (c) and both patching and decompression (a, b). (Constants: $h = 1$, $k_{\mathrm{err}} = 1$, $payload\_size = 114$, *buffer size* = 128). "Direct" shows the delay of transmitting the data directly, without processing.

BSDIFF with LZ77 or FastLZ, while for test cases 6 and 7, the lowest energy usage is registered using VCDIFF.

VCDIFF has good performance even without using an additional compressor. In fact, only FastLZ reduced the energy usage in all test cases. In the parameter change test cases, since the VCDIFF delta fits in one packet, there is no need to additionally compress it.

5) Delay: For reprogramming one node, using only compressed updates (Figure 6c) is much slower than sending the data directly. Slightly improved results are obtained when incremental updates are used - the processing time is larger than the transmission time. This is evident both for BSDIFF (Figure 6a) and VCDIFF (Figure 6b) in test cases 1 to 5. Using only VCDIFF is the best option in these cases.

In test cases 6 and 7, the processing overhead is significantly smaller compared to the transmission savings. Therefore, using LZ77, FastLZ or LZJB with BSDIFF, as well as only VCDIFF, is faster than transmitting the entire binary data.

If we vary any of the constants used in Figure 6, as shown in Figure 7, it is clear that compressed incremental updates are preferable to complete transfers in large networks.

C. Discussion

The presented results suggest that reprogramming can be improved in terms of energy efficiency and time required for update by using data compression and incremental updates. Improvements vary depending on the selection of algorithms.

Simply adding compression does not lead to lower energy usage or faster updates. In fact, some compression algorithms can degrade performance.

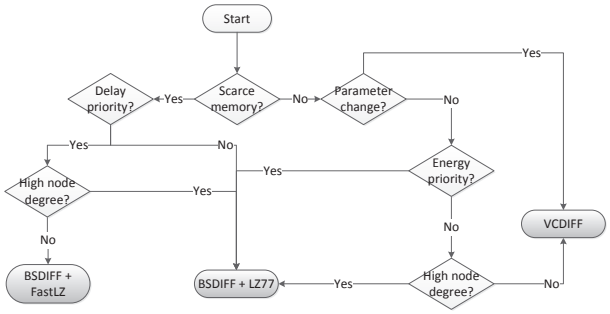
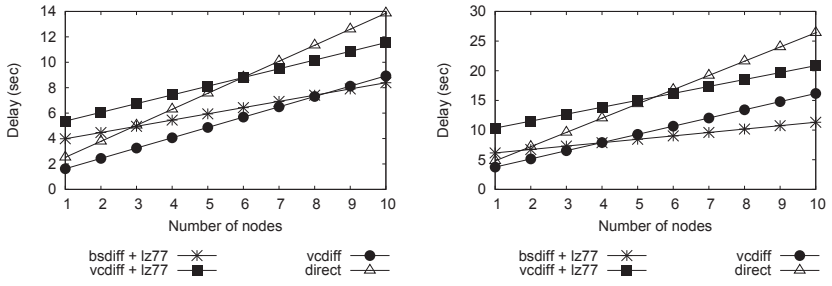


Fig. 8. Guidelines for selecting the best option for incremental update.

In contrast, using incremental updates showed solid results in all test cases. Up to 95% in energy savings were registered, along with 95% faster updates. Even though highest improvements were found during parameter reconfiguration, the fact that a 35% reduction in energy consumption was the minimum measured in specific configurations, gives strong arguments for using incremental updates in wireless sensor networks.

Selecting the best approach for incremental updates depends on the particular system. The four important factors that influence the selection are available resources, update type, network size and optimization goal (energy or delay). The choice is between using BSDIFF with either LZ77 or FastLZ, or using only VCDIFF. The decision tree, populated by recursively partitioning the gathered results, is shown in Figure 8.

The memory footprint of VCDIFF is a lot higher than BSDIFF with either LZ77 or FastLZ. Therefore, if resources



(a) Test case 1 (b) Test case 3

Fig. 7. Influence of number of nodes (h) on delay.

are scarce, VCDIFF is the least acceptable solution. On the other hand, VCDIFF has incomparable performance in terms of both energy usage and delay when parameter changes are considered. If most updates are of this type, then VCDIFF is the option to use. In case updates are more heterogeneous, the number of nodes in the network is small, and delay is a priority, then VCDIFF is again the best option. When energy usage is a priority, or the network is fairly large, then BSDIFF with LZ77 gives the best performance. Finally, when memory is scarce, the network is small and delay is a priority, then BSDIFF with FastLZ is the preferred option.

V. Conclusion

In this paper we investigated two approaches for efficient update distribution in networks of resource constrained devices. Firstly, we evaluated the performance of general purpose data compression algorithms applied directly on binary data. Secondly, we compared three algorithms for incremental update and combined them with the previously analyzed compression algorithms. Further tests were done on wireless sensor nodes, measuring memory requirements, code footprint, execution time, energy usage and delay.

Results show that data compression in combination with incremental update can significantly decrease energy usage and delay in reprogramming wireless sensor networks, but a bad choice can also increase it. The best option to perform incremental updates depends on multiple factors, for which we have provided a decision tree. Best performance was measured when using either the VCDIFF delta encoding algorithm, or the combination of BSDIFF for delta encoding and LZ77 or FastLZ for decompression.

Significant improvements can be reached by optimizing the incremental update process. One approach would be to adapt the original data in such way that it becomes as constant as possible between different versions. This can be achieved through function call indirection [8], extracting and ordering of global variables etc. As a result, the delta scripts would be much smaller, hence easier to compress.

References

[1] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th IEEE Int. Conf. on Local Computer Networks*, ser. LCN '04, 2004, pp. 455–462.

[2] I. Crossbow Technology, "Mote in-network programming user reference version 20030315," 2003.

[3] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *Symp. on Networked Systems Design and Implementation*, ser. NSDI, 2004, pp. 15–28.

[4] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Conf. on Architectural support for programming languages and operating systems*, ser. ASPLOS-X, 2002, pp. 85–95.

[5] R. Bosman, J. Lukkien, and R. Verhoeven, "An integral approach to programming sensor networks," in *Consumer Communications and Networking Conf.*, ser. CCNC, 2009, pp. 1–5.

[6] C. Miller and C. Poellabauer, "Reliable and efficient reprogramming in sensor networks," *ACM Trans. Sensor Networks*, vol. 7, 2010.

[7] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *Conf. on Sensor and Ad Hoc Communications and Networks*, ser. SECON, 2004, pp. 2–33.

[8] R. K. Panta, S. Bagchi, and S. P. Midkiff, "Efficient incremental code update for sensor networks," *ACM Trans. on Sensor Networks*, vol. 7, 2011.

[9] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *Conf. on Wireless sensor networks and applications*, ser. WSNA, 2003, pp. 60–67.

[10] F. Marcelloni and M. Vecchio, "An efficient lossless compression algorithm for tiny nodes of monitoring wireless sensor networks," *The Computer Journal*, vol. 52, no. 8, pp. 969–987, 2009.

[11] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Conf. on Embedded networked sensor systems*, ser. SenSys, 2006, pp. 265–278.

[12] K. Dolfus and T. Braun, "An evaluation of compression schemes for wireless networks," in *Cong. on Ultra Modern Telecommunications and Control Systems and Workshops*, ser. ICUMT, 2010, pp. 1183–1188.

[13] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient sensor network reprogramming through compression of executable modules," in *Conf. on Sensor, Mesh and Ad Hoc Communications and Networks*, ser. SECON, 2008, pp. 359–367.

[14] M. Stolikj, P. J. Cuijpers, and J. J. Lukkien, "Energy-aware reprogramming of sensor networks using incremental update and compression," *Procedia Computer Science*, vol. 10, pp. 179–187, 2012.

[15] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University, 1999.

[16] D. Korn, J. MacDonald, J. Mogul, and K. Vo, "The VCDIFF Generic Differencing and Compression Data Format," RFC 3284, IETF, 2002.

[17] J. Macdonald, "Xdelta - open-source binary diff," 2011. [Online]. Available: http://www.xdelta.org/

[18] C. Percival, "Naive differences of executable code," 2003. [Online]. Available: http://www.daemonology.net/bsdiff/

[19] D. Albu, J. Lukkien, and R. Verhoeven, "Energy effect of on-node processing of ecg signals," in *Conf. on Consumer Electronics*, ser. ICCE, 2010, pp. 7–8.

[20] Jennic, "Calculating 802.15.4 data rates," Appl. note JN-AN-1035, 2006.

[21] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Symp. on Information processing in sensor networks*, ser. IPSN, 2005.

[22] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt, "Enabling Large-Scale Storage in Sensor Networks with the Coffee File System," in *Conf. on Information Processing in Sen. Netw.*, ser. IPSN, 2009, pp. 349–360.