FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks

Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel

IPVS, Universität Stuttgart
Universitätsstr. 38
D-70569 Stuttgart, Germany
{`marron, gauger, lachenmann, minder, saukh, rothermel`}`@informatik.uni-stuttgart.de`

Abstract. The ability to update the program code installed on wireless sensor nodes plays an import role in the highly dynamic environments sensor networks are often deployed in. Such code update mechanisms should support flexible reconfiguration and adaptation of the sensor nodes but should also operate in an energy and time efficient manner. In this paper, we present `FlexCup`, a flexible code update mechanism that minimizes the energy consumed on each sensor node for the installation of arbitrary code changes. We describe two different versions of `FlexCup` and show, using a precise hardware emulator, that our mechanism is able to perform updates up to 8 times faster than related code update algorithms found in the literature, while consuming only an eighth of the energy.

1 Introduction

The continuous miniaturization process of computing devices combined with the proliferation of sensor technology has led to an increase in the number and the variety of devices that are able to sense their environment, gather and process data and communicate their results either to a base station or to other neighboring devices. Such wireless sensor networks are usually characterized by the limited resources available at each individual device, and the fact that each sensor node cooperates with its peers in a distributed fashion to accomplish a common task.

The ability to update the program code installed on wireless sensor nodes is an important feature in such systems, necessary not only for correcting errors but also for being able to adapt the running software to changed environmental conditions or modified application requirements. In particular, we expect a growing demand for adaptive system software support in sensor networks due to the increasing complexity of applications and the inherent dynamics of typical sensor network environments.

Current system software implementations do not provide the flexibility needed to dynamically adapt the software running on sensor nodes. This motivates the work of the `TinyCubus` project in our research group [1], which aims at developing a generic and reconfigurable system software for sensor networks based on `TinyOS`. Two important building blocks of `TinyCubus` are support for structured cross-layer optimizations provided by the `Tiny Cross-Layer Framework` and adaptation capabilities for system and application components provided by the `Tiny Data Management Framework`.

In this paper, we present `FlexCup` ("FLEXible Code UPdates"), a code update mechanism that enables on the fly reinstallation of software components in `TinyOS`-based sensor nodes in an efficient way. `FlexCup` has been developed as part of `TinyCubus` to provide code update capabilities for the adaptation of components in the `Tiny Data Management Framework`, but it can also be used independently as a general code update mechanism for sensor networks.

A code update mechanism for sensor networks needs to take into account that sensor networks usually consist of small devices with extreme resource limitations. The optimization of resource usage and energy considerations are therefore crucial challenges that needed to be addressed in the development of `FlexCup`.

The mode of operation of `FlexCup` is divided in two phases: the code generation phase, where relevant information is generated at compile time; and the linking phase, where the modified components are combined with other components at runtime.

The remainder of this paper is structured as follows. Section 2 starts by giving information about existing approaches and their shortcomings. Section 3 deals with the details of our code update approach and its compile time and runtime algorithms. Section 4 gives experimental results on the complexity and efficiency of `FlexCup` by comparing it to other approaches. Finally, section 5 concludes this paper and gives some insight regarding future work.

2 Related Work

`TinyOS` [2] is probably the most widely-used operating system for sensor networks and is the target system for `FlexCup`. It has been ported to several hardware platforms including the MICA2 motes from Crossbow Technologies. Thanks to the wiring and event abstractions available in `nesC`, the component-based programming abstraction for `TinyOS` [3], this operating system is well suited for the requirements of sensor network applications. However, `TinyOS` does not allow components to be replaced at runtime. Instead, the entire program image containing both system and application components has to be exchanged if any one of the components needs to be replaced.

Another operating system developed for sensor networks and other resource constrained devices is Contiki [4]. In contrast to `TinyOS`, Contiki does provide support for dynamic loading of applications and system services as a core functionality of the system. However, this flexibility requires additional levels of indirection for calls to these dynamic services which adds some runtime overhead.

Maté [5] executes application code using a virtual machine. Since the actual application is only stored in RAM, this system can easily deal with code updates, but the overhead of running a virtual machine on each sensor node is considerable. The advantage of virtual machine approaches is that the size of their bytecode can be smaller than native code, which reduces the energy consumed for code transfers. However, for long-running applications the energy overhead generated by code interpreted at runtime outweighs this advantage [5].

There are several middleware solutions for sensor networks that provide some functionality related to our work. For example, Impala [6] adds abstractions that allow for dynamic updates of modules and adaptation. Modules which are already linked and which are reused in a new software version do not have to be re-linked. However, both the update and the adaption mechanism have not been implemented in the actual system yet. MiLAN [7] monitors the network situation and manages the quality needs of applications by adapting its behavior and optimizing the network topology. While MiLAN is in that way able to change its operation at runtime, it does not include support for dynamic code updates.

Several approaches try to efficiently disseminate code updates in a sensor network using multi-hop communication [8, 1, 5, 9, 6]. However, the efficient distribution of code is an issue orthogonal to our approach, which aims at reducing the size of such updates and adding the flexibility needed for adaptation. With little effort, `FlexCup` can be combined with any of these approaches.

Many current algorithms dealing with code updates always transmit the complete code image (including the system software), which usually amounts to several kilobytes of data. One example of this approach is `Deluge` [8]. `Deluge` is included in recent `TinyOS` releases and provides functionality to disseminate code updates in multi-hop networks while keeping the number of network packets low.

A more advanced approach found in the literature to reduce the number of packets to be transmitted for each code update is to compare the new code with the previously installed software version and transmit only the differences. Reijers and Langendoen [10] use a diff-like approach to compute a diff script that transforms the installed code image into a new one. Likewise, the incremental network programming protocol presented by Jeong and Culler [11] uses the Rsync algorithm [12] to find variable-sized blocks that exist in both code images and then only transmits the differences. However, both of these approaches just compare the code image using very limited knowledge about the application structure, if at all.

Koshy and Pandey [13] describe a scheme that uses incremental linking (on a PC) to reduce the number of changes in the code and transmit the code update with a diff-like algorithm. They leave most parts of the previous program image unchanged and modify only those functions that actually change. In order to avoid address shifts when the size of a function changes they add empty space behind each function. However, this approach does not provide the flexibility offered by `FlexCup` since the linking process is still performed at the base station. Koshy and Pandey even argue that linking on the sensor nodes – the very approach of `FlexCup` presented in this paper – cannot be done in practice due to high costs for transmission and storage of object files.

Most approaches assume that code updates will be distributed to all nodes in the network. However, the complexity of applications and the need for reconfiguration indicate that it might be desirable to install the required components only on those nodes that need it and maybe store other components in a free part of flash memory for later adaptation, since flash memory is typically less limited than program memory. Therefore, our solution uses knowledge about the application structure by grouping the components forming the application and the operating system. It offers more flexibility than just replacing arbitrary pieces of code because it makes it possible to dynamically change the current set of installed components through adaptation. That way, the sensor nodes can store several instances of a component at a time even though they only need one of them to fulfill their current task. When the task changes or other factors make it necessary, the node can easily replace the currently used component with a more efficient one.

3 FlexCup

`FlexCup` implements an efficient code update algorithm that allows exchanging only the components of a program that have actually changed. This helps saving deployment time as well as energy on the sensor nodes.

To perform its tasks, `FlexCup` needs to be involved in the process of compiling the components on the base station, and installing the code update on the sensor nodes: During the code generation process, `FlexCup` generates meta-data that describes the compiled components. `FlexCup` then uses this meta-data during a code update to place the new component inside the running application, relink function calls to the appropriate locations and perform address binding of data objects, as we will see in the next sections.

Using this method, `FlexCup` is able to reconfigure, exchange or reinstall parts of an application running on sensor nodes without having to retransmit the whole program image. Furthermore, since there is no real distinction between system and application components in current sensor network devices, `FlexCup` can be used for updating parts of `TinyOS` or `TinyCubus` just as it can be used for exchanging application components.

We have implemented and tested `FlexCup` using the MICA2 sensor platform available from Crossbow Technologies. Although developed as part of `TinyCubus`, the implementation of `FlexCup` is independent of any operating system or system software. This has two advantages: First, since `FlexCup` is written in ANSI C and does not have dependencies to specific system libraries, it can be easily ported to other frameworks. Second, `FlexCup` only runs during the process of installing code updates and does not impose further restrictions on the RAM available for application components.

3.1 Component and Meta-Data Generation

Component Generation `TinyOS` applications developed using the `nesC` programming language consist of a set of system and application components that are "wired" to generate a running program. The `nesC` compiler produces a single C file combining the source code of all these components, thereby generating a tightly interwoven application. This approach has the advantage that the compiler can perform optimizations like function inlining on the entire program. However, there is no simple way of replacing only a part of the compiled program like exchanging a component or a function inside a component.

This potential limitation is to some extent addressed by a new concept introduced in a recently released version of the `nesC` compiler (`nesC` 1.2). The new concept allows compiling a set of `nesC` components into a separate object file, a so called *binary component*. Such binary components can be wired like traditional `nesC` components and are then combined by the linker to create the complete application code. However, this linking is still done on the base station prior to the deployment in the sensor network.

`FlexCup` uses the concept of binary components and extends it by performing the linking process on the sensor node itself.

The use of binary components still allows the compiler to perform code optimizations inside of the individual binary components. Global optimizations are no longer possible. However, if the application developer segments the application based on the semantic relation of the components, we expect an application using binary components to perform similarly to a globally optimized version.

Our experience with `FlexCup` indicates that a reasonable segmentation of an application into binary components can be easily identified by examining the semantics of the components and their use in the system. We used several heuristics including the degree of interaction with other components and the expected likelihood of components being exchanged together. Typical examples of components combined into individual binary components are the ones implementing radio communication, the sensor access and the application components.

The segmentation of components into binary components is a design decision to be made by the system and application developers. In the long run, we expect most system components of `TinyOS` to be available segmented into binary components, so that the application developer will only have to consider his own components implementing application-specific functionality.

Meta-Data Generation `FlexCup` requires meta-data to be able to integrate new components into the existing program code on a sensor node. On the sensor nodes, this meta-data is stored in external flash memory and consists of the following three parts: generic *program information*, a program-wide *symbol table*, and a *relocation table* for each binary component in the program. The generic program information lists the number and relative offsets of all binary components, as well as the addresses of the symbol and relocation tables. The symbol table contains information about the global data and function symbols used in all components, sorted by their identifiers in ascending order. The relocation tables list the references from inside the component code to data or function symbols specified in the symbol table. Fig. 1 shows a pictorial representation of a sample program consisting of three binary components and its representation in the external flash memory after being loaded onto a sensor node.

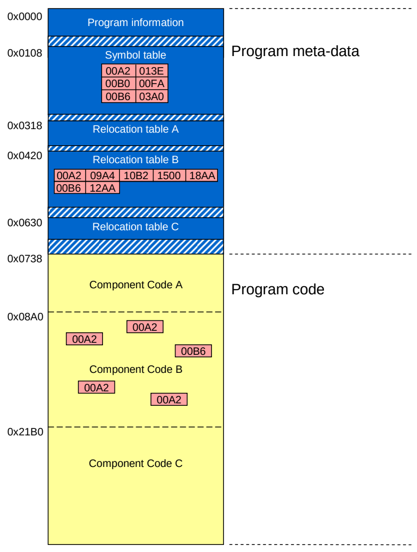

Fig. 1. Sample code and meta-data

As can be seen in Fig. 1, `FlexCup` stores a copy of the program code in the external flash right after the meta-data. This copy is used for constructing the new program code during a code update. Our implementation leaves free spaces (hatched blocks in Fig. 1) between the symbol table, each relocation table and the program code to allow for size changes of this data without having to pay the penalty of moving large pieces of data or even the whole program code.

Optimizations The transmission and storage of the meta-data required for the dynamic linking of the components incurs an overhead on the sensor nodes. We have implemented several optimizations to minimize these effects: First, symbols in the symbol and relocation tables are identified by a two-byte id instead of a human-readable string. Second, we compress the size of the relocation tables by combining entries with the same id. For example, if there are several relocation entries referencing the same symbol, all entries are grouped together so that the identifier is needed only once. These simple optimizations incur savings in space of over 40% with respect to our original implementation.

3.2 Runtime Linking

Fig. 2 outlines the sequence of operations performed by `FlexCup` on a sensor node during the update of a binary component. The operation of the algorithm can be split up into five steps: (1) Storage of code and meta-data; (2) symbol table merge; (3) relocation table replacement; (4) reference patching; (5) installation and reboot.

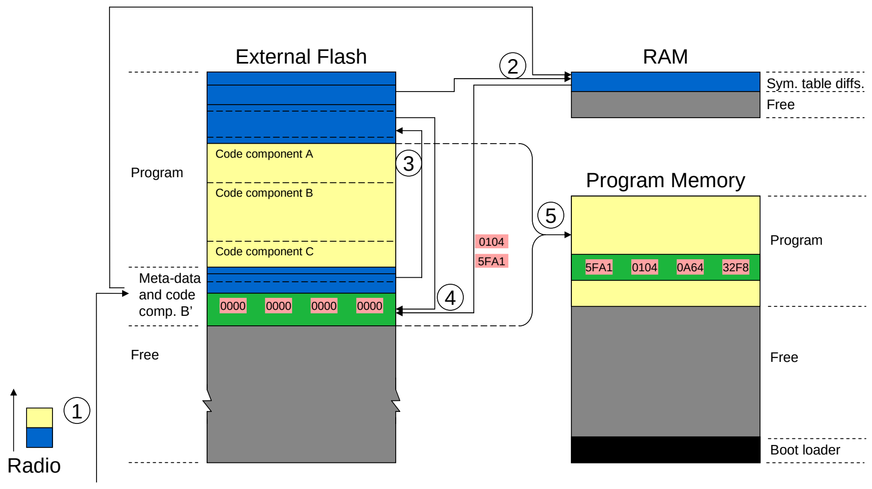

Fig. 2. Runtime linking process

(1) Storage of Code and Meta-Data The first step in the runtime linking process involves receiving the update data, including code and the meta-data of the component, and storing it into flash memory. The external flash memory of the MICA2 sensor nodes has a capacity of 512 kBytes and is normally used to store sensor readings and measurement results. For the purposes of `FlexCup`, flash memory is used as an external memory component where code updates and program meta-data can be stored for processing.

Even though accesses to the flash memory are very costly, using the flash memory for storing the received data is necessary for two reasons: First, the size of a code image is usually much larger than the 4 kBytes of RAM available on the nodes, so that the code image cannot be prepared completely in RAM. Second, the program code can only be written to program memory from a special bootloader section and writes are only possible on a page by page basis, so that the code image must be prepared externally before writing it to program memory. However, the degree of use of the external flash memory directly influences the amount of energy consumed by the algorithm.

Regarding the actual transmission of the modified binary component and meta-data, `FlexCup` allows two different modes of operation. The first one, called `FlexCup Basic`, transfers the whole binary component and its meta-data without considering the data already stored on the sensor node. This algorithm can be inefficient, especially if the binary component is relatively large and the number of changes is small. For this reason, `FlexCup` also supports a diff-based approach, called `FlexCup Diff`, that only transfers the incremental changes between the new binary component and the one already stored on the sensor node. `FlexCup Diff` can operate more efficiently than pure diff-based solutions as the processed binary code does not yet contain references to specific addresses in memory (only default values). For this reason, address shifts, which are one of the main reasons for change entries in pure diff-based approaches, do not increase the size of the data transmitted. However, just like in all diff-based approaches, the base station needs to have knowledge about the exact configuration of the sensor nodes in order to be able to prepare the diff script.

(2) Symbol Table Merge The second step of the linking process involves combining the existing program symbol table with the newly received symbol data. Since both tables are sorted by symbol id in ascending order, an algorithm similar to merge sort is used to create the new symbol table.

Merging is performed with the help of 3 kBytes of temporary buffer in RAM used by `FlexCup` to store all changed symbol information. This buffer space is guaranteed to be available since `FlexCup` does not run in parallel to the application[1]. The advantage of this buffering is that accessing RAM is much faster and much more energy efficient than using the external flash for all operations. At the end of this step, the new symbol table is written back to the external flash at once.

A challenging task for `FlexCup` is the management of the application's data variables. Each component uses a set of variables, initializing some of them with predefined values. `FlexCup` has to cope with possible size changes, changed initialization data and the addition and the removal of such variables. It needs to calculate an adequate layout for the storage of the variables in data memory and needs to set the symbol addresses accordingly. `FlexCup` also has to prepare the initial values that are then loaded during the startup of the system.

(3) Relocation Table Replacement This step deals with the replacement of the relocation table. This task is much simpler than the previous step, because each binary component contains an individual relocation table sent as part of the component update. Correspondingly, this step only involves copying the new relocation table to the appropriate location and, if necessary, shifting the following tables backward by the right amount of bytes.

(4) Reference Patching The fourth step involves going through the entries of the relocation tables of all components, and checking whether any of the references needs to be updated. An update is required for all references coming from the new component code and for all references to symbols that changed their destination address during the update. If an update is required, `FlexCup` jumps

[1] It is not possible to use all 4 kBytes of RAM for the symbol table because `FlexCup` itself needs 724 bytes of RAM for its operation.

to the address specified in the relocation table and writes the new destination address value. This procedure strongly benefits from the fact that the change set of entries in the symbol table is already buffered in RAM and does not need to be searched for again in flash memory. At the end of the reference patching step, all references of the components point to the right location in program or data memory and the code image is ready to be copied to program memory.

(5) Installation and Reboot The last step takes care of copying the program code from external flash to program memory and reboots the sensor node afterwards. This is done using a custom bootloader installed in the bootloader section of the processor's program memory.

One important reason for rebooting the sensor nodes are potential layout changes of the application. Without a reboot, pointer variables might point to locations in memory that are no longer valid. If the sensor network application needs to preserve state despite a reboot, it is necessary to use an external mechanism that saves the application state to non-volatile memory.

4 Experimental Evaluation

To evaluate the performance of `FlexCup` in terms of time and energy consumed for the update of sensor network applications, we compare `FlexCup Basic` and `FlexCup Diff` with two related approaches found in the literature: `Deluge` [8] and a diff-based update mechanism (from now on "`MOAP-Diff`") available as part of the MOAP project [9, 14]. `Deluge` transmits the whole program image as a monolithic block of code, whereas `MOAP-Diff` implements a modified version of Reijers and Langendoen's original diff algorithm.

4.1 Experimental Setup

For the performance measurements detailed below, we have used a modified version of the MICA2 emulator `atemu` [15] which we calibrated using measurements on real sensor hardware. The modified version of the emulator includes an implementation of the external flash memory component found on the MICA2 sensor nodes and allows precise measurements of the energy consumption and the time needed to run the algorithms under test. The experimental results have been obtained by calculating the average results over 20 runs.

Selected Applications For our performance comparisons we have selected three typical applications that can be downloaded from the `TinyOS` CVS repository:[2] `OscilloscopeRF`, `Surge` and `AcousticLocalization`. `OscilloscopeRF` is a simple application that periodically reads a sensor value and transmits it via radio to a base station located within transmission range. `Surge` is similar

[2] `http://cvs.sourceforge.net/viewcvs.py/tinyos/`

to OscilloscopeRF, but includes a multi-hop routing protocol that dynamically builds a routing tree along which sensor readings are forwarded to the base station. Finally, AcousticLocalization is able to determine the distance of neighboring sensor nodes by taking advantage of the difference in speed of radio waves and sound. Table 1 gives details about the complexity of the three applications showing the respective code size, the number of nesC components and the number of binary components.

Table 1. Complexity of sample applications

Applications	**Size (bytes)**	**Number of nesC components**	**Number of binary components**
OscilloscopeRF	11784	39	6
Surge	17096	53	10
AcousticLocalization	24272	69	15

Code Modifications Using the described applications as test cases for the code update algorithms, we examine three different classes of changes to the code, ranging from small updates or bug fixes through internal updates affecting only a single binary component to external changes that imply the modification and update of several binary components at the same time.

Table 2. Changes performed on the applications

Application	**Class**	**Code Update**
OscilloscopeRF	small	global constant
OscilloscopeRF	small	additional call
OscilloscopeRF	small	sensor reading
Surge	internal	function exchange
Surge	internal	wiring configuration
AcousticLocalization	external	component exchange

Table 2 gives an overview of the changes we have performed for the experiments below, as well as the class they belong to. The three modifications to the OscilloscopeRF application are relatively simple. They involve changes to the port data is sent to (global constant), the addition of a call to an initialization function (additional call), and a modification of the value returned by the sensor (sensor reading).

The two internal modifications to Surge involve, in the first case, the replacement of the shortest-path-first routing algorithm with MintRoute, another routing algorithm providing the same interface that considers the quality of links for route selection. The second change involves the removal of the LED interface used for visual feedback which causes changes to the wiring configuration of the application. Finally, our last and only external modification changes the routing algorithm in the `AcousticLocalization` application to disallow the forwarding of messages – changing the modified nodes to behave as leaf nodes.

4.2 Size of Components and Meta-Data

The first characteristic that distinguishes one code update algorithm from another is the amount of code and meta-data involved in the process of a code update installation. We consider two different metrics: (1) the size of the code update algorithm itself, and (2) the amount of data transmitted over radio for each update. For the evaluation of the second metric, we assume that both the original application and the code update algorithm have already been installed in program memory. Therefore, the sensor node is able to receive the code update and, depending on the algorithm, process the code image (`Deluge`), interpret the diff script (`MOAP-Diff`), or perform the linking process (`FlexCup Basic` and `FlexCup Diff`).

Table 3. Average size of code update algorithms

	Program Code Size (bytes)		
Application	`Deluge`	`MOAP-Diff`	`FlexCup`
`OscilloscopeRF`	10868	16742	26715
`Surge`	11326	17213	27466
`AcousticLocalization`	10650	16728	26692

Table 4. Size of components and meta-data (in bytes)

	Transmitted Data Size								Flash Memory Data Size			
		`MOAP-`	`FlexCup Basic`			`FlexCup Diff`				`MOAP-`	`FlexCup`	
Code Update	`Deluge`	`Diff`	**Meta**	**Code**	**Total**	**Meta**	**Code**	**Total**	`Deluge`	`Diff`	`Basic`	`Diff`
global const.	23142	11	799	1198	1997	530	15	545	23142	28538	37337	35885
additional call	23142	1230	801	1202	2003	760	5	765	23142	28542	37343	36105
sensor reading	23142	2835	537	886	1423	523	114	637	23142	28608	36743	35977
function exch.	28652	7684	1056	3258	4314	1110	1587	2697	28652	33440	43561	41944
wiring config.	28652	375	1355	2142	3497	1290	8	1298	28652	34272	42744	40545
comp. exch.	34162	7802	2565	4773	7338	2611	532	3143	34162	40156	58014	53736

Table 3 shows the average size of the three code update algorithms we evaluate in this paper as they are installed in program memory. `MOAP-Diff` is about 55% larger than `Deluge`, and `FlexCup` is in turn about 60% larger than `MOAP-Diff`. For all three algorithms, the exact sizes differ between applications because they have to be compiled together with the application code. The resulting size differences are due to differences in the set of system components already included by the applications and to different optimizations performed by the `nesC` compiler. In the case of `FlexCup`, however, there is a fixed part of 16212 bytes that is compiled and executed independently of the application and is, therefore, not subject to these effects. As program memory size does not seem to be a limiting factor for most current sensor network applications, we do not expect these differences in code size to inhibit the use of `FlexCup`.

Apart from the size of the code update algorithms and the one-time penalty that sensor nodes have to pay for their installation, a more relevant metric is the amount of data to be transmitted when an application is modified. The left-hand side of Table 4 shows the number of bytes transmitted by `Deluge`, `MOAP-Diff`, `FlexCup Basic` and `FlexCup Diff` for performing the six code updates introduced in the previous section. For example, for the modification of the `OscilloscopeRF` application so that it returns a different sensor reading (third code update in Table 4), `Deluge` has to transmit 23142 bytes, whereas `MOAP-Diff` only requires transmitting 2835 bytes. As detailed in section 3, both `FlexCup Basic` and `FlexCup Diff` need to transmit meta-data, i.e., the symbol and relocation tables, as well as the code of the component that changes. In total, `FlexCup Basic` transmits 1423 bytes and `FlexCup Diff` only 637 bytes. This implies more than 90% savings in the number of packets comparing `FlexCup Basic` to `Deluge` and more than 75% if we compare `FlexCup Diff` to `MOAP-Diff`.

Only in cases where the actual change is very small, like for the change of a global constant in `OscilloscopeRF` or the update of the wiring configuration in `Surge`, does `MOAP-Diff` perform much better than any other algorithm under test. For `FlexCup Diff`, much of the overhead comes from the transmission of the new meta-data information. Looking at the pure code size, `FlexCup Diff` would easily outperform `MOAP-Diff` in most cases. In general, `FlexCup Basic` only requires between 6% and 21% of the number of transmissions of `Deluge`. `FlexCup Diff` saves more than 75% in the best case compared to `MOAP-Diff` and requires less transmission volume than `MOAP-Diff` in four out of six cases.

However, there is another relevant factor. The right part of Table 4 shows the size of the information that needs to be stored in flash memory for performing a given update. In the case of `Deluge`, this is just the data received over the radio link, but for `MOAP-Diff` and `FlexCup` this also includes the code of already installed components and meta-data. For that reason, `Deluge` is in all cases more efficient than the three other approaches in terms of space complexity. This might be an important factor in scenarios where the application stores large amounts of data locally on the sensor nodes before uploading it to a base station. Although the size of the flash memory on the MICA2 nodes amounts to 512 kBytes and should be sufficient for most applications, long-running applications might still benefit from the extra space. In general, `MOAP-Diff` requires between 16 and 25% more space than `Deluge`. `FlexCup Diff` needs about 50% more space and the space requirements of `FlexCup Basic` are 50 to 70% higher than the requirements of `Deluge` depending on the specific code update. Most of the overhead of the last two approaches comes from the storage of the symbol and relocation tables.

4.3 Efficiency of the Code Update Algorithms

Let us now look at the performance characteristics of the four code update algorithms. For the purpose of the experiments below, we measure the efficiency of the algorithms based on two metrics: execution time and energy consumption.

In general, it is clear that both metrics are not independent from each other and that a longer execution time usually also implies a higher energy consumption. However, the diverse energy characteristics of the hardware components, especially the external flash memory and the radio interface, indicate that considering the execution time alone is not sufficient for determining which algorithm is more energy-efficient.

In the experiments, we consider execution time and energy consumption throughout the three phases "Transmission", "Processing" and "Installation". To guarantee a fair comparison between the algorithms, we assume perfect conditions during the transmission phase, ignoring possible collisions, errors and the specific protocol overhead. To simplify the evaluation, we also assume that each node receives the code update data exactly once and then forwards it to its neighbor nodes.

Execution Time Fig. 3 shows the duration of the three phases of the four code update mechanisms for all six code updates. As one can see, the time required for installation is dominated by the transmission time in the case of `Deluge` and by the processing time for `MOAP-Diff`, whereas in the case of `FlexCup Basic` and `FlexCup Diff`, transmission and processing times are of similar magnitude.

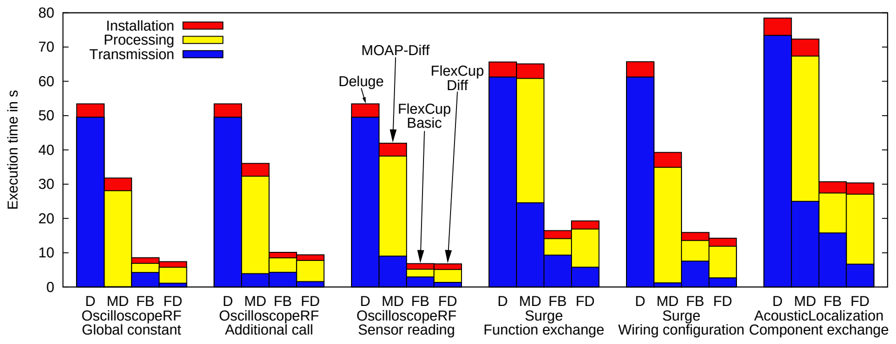

Fig. 3. Execution times of the code updates

`FlexCup Basic` and `FlexCup Diff` are about 4 times faster than `MOAP-Diff` and have on average almost 8 times better execution times than those of `Deluge`. `FlexCup Diff` is marginally faster than `FlexCup Basic` by saving a significant part of the transmission time while requiring some additional time for processing the diff script.

In general, MOAP-Diff is faster than Deluge, although large parts of its savings in transmission time are spent for processing the diff script in the processing phase. An extreme example can be seen in the 'function exchange' case of the Surge application, where the difference in execution time between MOAP-Diff and Deluge amounts to only 0.55 seconds. FlexCup Diff is actually slower than FlexCup Basic in this example with the processing effort outweighing the transmission savings.

Energy Consumption Fig. 4 shows the amount of energy in millijoules consumed by the four code update mechanisms during the three phases of the code updates. The measurements confirm the good performance of FlexCup compared to the other approaches, showing that in the best case FlexCup consumes only an eighth of the energy of Deluge and MOAP-Diff. On the other hand, the results cannot confirm the relatively good performance of MOAP-Diff compared to Deluge observed for the execution times of the code updates. Energy-wise, MOAP-Diff performs worse than Deluge in three out of the six cases. This is mainly due to the relatively inefficient implementation of MOAP-Diff which uses a lot of access operations to the external flash memory. These access operations are very energy expensive on the MICA2 hardware.

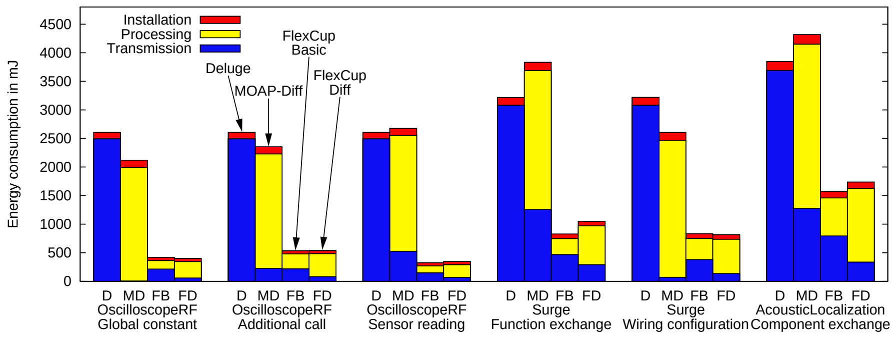

Fig. 4. Energy consumption of the code updates

The expensive implementation of MOAP-Diff also explains the inferior performance of FlexCup Diff, which uses the MOAP-Diff algorithm for extracting its component data. It remains to be investigated whether a more efficient implementation of the diff approach is able to retain more of the advantage achieved during the transmission phase and to improve on the results of FlexCup Basic.

Nevertheless, our experiments show that FlexCup Basic and FlexCup Diff, although similar in energy consumption and execution time, use the sensor node hardware in different ways. In general, FlexCup Basic transmits more data than FlexCup Diff, but the latter has extra overhead regarding the decoding of the binary component to be installed in program memory. Thus, depending on the physical characteristics of the external flash memory and radio components, it might be preferable to use `FlexCup Basic` instead of `FlexCup Diff`, or vice versa.

An additional lesson that can be learned from the results in Fig. 3 and 4 is that counting the number of bytes a code update algorithm needs to transmit does not necessarily give information about the time and energy efficiency of the algorithms. All relevant factors, including processing and flash memory access costs, need to be part of the evaluation.

4.4 Advantages and Limitations of FlexCup

`FlexCup` exhibits several advantages compared to other code update mechanisms. First of all, it allows for greater *flexibility* in the exchange of application and system software components at runtime, thereby offering functionality required by *adaptive system software* like `TinyCubus`. Second, `FlexCup` is able to reduce the number of bytes transferred to each sensor node and to minimize the amount of energy needed for the processing of code updates, which immediately translates into a *better overall energy consumption*.

One limitation of `FlexCup` is its use of external flash memory for the storage of meta-data and the use of program memory for the storage of the `FlexCup` program code. Both are only possible if there is enough free space available after fulfilling the requirements of the application. Like other code update mechanisms, `FlexCup` also has to deal with the access characteristics of the platform's flash memory. Especially the problem of wear levelling in flash memory remains to be addressed.

5 Conclusions and Future Work

In this paper we have presented `FlexCup` a flexible code update mechanism for sensor networks that offers the functionality and performance required by adaptive system software. We have evaluated `FlexCup` by analyzing several realistic code updates with the help of emulation tools calibrated on real sensor nodes. Compared to related approaches, `FlexCup` was able to perform the same updates up to 8 times faster while consuming only an eighth of the energy.

We have also shown that the overall code image size of `TinyCubus` and `FlexCup`, as needed for the reconfiguration functionality required by more complex sensor network applications, is comparable to other approaches such as `Deluge` and Reijers' and Langendoen's diff-based algorithm, although `FlexCup` is able to provide more flexibility and adaptation capabilities.

Regarding future work, we would like to explore more complex algorithms for the management of flash memory and reserved RAM space to further reduce the time and energy consumption for linking in `FlexCup`. We are also considering the use of more efficient diff algorithms that would contribute to reducing the amount of energy needed for the execution of the diff scripts in `FlexCup Diff`. Finally, it would also be interesting to evaluate the influence of different hardware properties on our implementation by porting `FlexCup` to other platforms such as the EYES sensor nodes.

References

1. Marrón, P.J., Lachenmann, A., Minder, D., Hähner, J., Sauter, R., Rothermel, K.: TinyCubus: A flexible and adaptive framework for sensor networks. In: Proc. of the 2nd European Workshop on Wireless Sensor Networks. (2005) 278–289
2. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems. (2000) 93–104
3. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation. (2003) 1–11
4. Dunkels, A., Grönvall, B., Voigt, T.: Contiki – a lightweight and flexible operating system for tiny networked sensors. In: Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I). (2004)
5. Levis, P., Culler, D.: Maté: A tiny virtual machine for sensor networks. In: Proc. of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems. (2002) 85–95
6. Liu, T., Martonosi, M.: Impala: A middleware system for managing autonomic, parallel sensor systems. In: Proc. of the 9th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming. (2003) 107–118
7. Heinzelman, W.B., Murphy, A.L., Carvalho, H.S., Perillo, M.A.: Middleware to support sensor network applications. IEEE Network **18** (2004) 6–14
8. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proc. of the 2nd Intl. Conf. on Embedded Networked Sensor Systems. (2004) 81–94
9. Stathopoulos, T., Heidemann, J., Estrin, D.: A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, L.A. (2003)
10. Reijers, N., Langendoen, K.: Efficient code distribution in wireless sensor networks. In: Proc. of the 2nd ACM Intl. Conf. on Wireless Sensor Networks and Appl. (2003) 60–67
11. Jeong, J., Culler, D.: Incremental network programming for wireless sensors. In: First IEEE Comm. Soc. Conf. on Sensor and Ad Hoc Communications and Networks. (2004)
12. Tridgell, A.: Efficient Algorithms for Sorting and Synchronization. PhD thesis, The Australian National University (1999)
13. Koshy, J., Pandey, R.: Remote incremental linking for energy-efficient reprogramming of sensor networks. In: Proc. of the 2nd European Workshop on Wireless Sensor Networks. (2005) 354–365
14. Yeh, T., Yamamoto, H., Stathopolous, T.: Over-the-air reprogramming of wireless sensor nodes. UCLA EE202A Project Report (2003) `http://lecs.cs.ucla.edu/~thanos/EE202a_final_writeup.pdf`.
15. Polley, J., Blazakis, D., McGee, J., Rusk, D., Baras, J.S.: ATEMU: a fine-grained sensor network simulator. In: Proc. of the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks. (2004)