ARTICLE IN PRESS

Ad Hoc Networks xxx (2012) xxx–xxx

Contents lists available at SciVerse ScienceDirect

Ad Hoc Networks

journal homepage: www.elsevier.com/locate/adhoc

Enabling efficient reprogramming through reduction of executable modules in networked embedded systems

Wei Dong *, Chun Chen, Jiajun Bu, Chao Huang

Zhejiang Provincial Key Laboratory of Service Robot, College of Computer Science, Zhejiang University, Hangzhou 310027, PR China

ARTICLE INFO

Article history:
Received 18 January 2012
Received in revised form 25 July 2012
Accepted 26 July 2012
Available online xxxx

Keywords:
Modules and interfaces
Loaders
Embedded systems
Wireless sensor network

ABSTRACT

We present a systematic modular design approach for networked embedded systems. We effectively reduce the module file size to enable efficient network reprogramming, while at the same time retain necessary information to maintain module flexibility. We further handle module dependencies in a fine-grained manner, which improves system reliability while keeping the system configuration to its minimum requirement. We have implemented the modular approach based on a micro embedded OS, SenSpire OS, for AVR and MSP430 platforms. The evaluation results show that the proposed SELF module file format is 4.6–7.6 times smaller than the standard ELF format, and is 1.6–2.4 times smaller than the CELF format (a Compact ELF format for the Contiki OS). SELF retains necessary information to enable flexible modular programming and inter-module communications. We have further developed a long-term energy efficiency model to explore the tradeoffs of different reprogramming approaches.

1. Introduction

Networked embedded systems, such as wireless sensor networks (WSNs), have been proposed for a wide range of applications such as military surveillance, habitat monitoring, and infrastructure protection [1,2]. WSNs are also deployed in many industrial environments for sophisticated sensing and control purpose [3]. In industrial monitoring applications, WSNs can be used to remotely monitor the health of equipments and machineries, allowing users to detect machine failures and to reduce repair cost. WSNs are essential to industrial automation and management. In inventory management systems, WSNs improve the visibility of materials and enable users to manage and control real-time inventory data.

In many industrial environments, we usually experience a great number of hazards that can range from strong mechanical vibrations, high temperatures, fragile surfaces, and even explosive gases. It is difficult or infeasible to physically collect previously deployed nodes. On the hand, WSN software often needs to be changed after deployment for a variety of reasons—upgrading node software, correcting software bugs, and patching security holes. Enabling sensor nodes to be reprogrammable over the air is crucial for the maintenance of WSNs [4,5].

In the reprogramming process, the loading mechanism of a sensor node is responsible to load a new code image (disseminated over the air) onto the program flash, enabling sensor nodes to execute the new code. The loading mechanism impacts the overall reprogramming efficiency: a simple bootloader, e.g., `TOSBoot` [6], requires the replacement of an entire application image, which is not energy-efficient to disseminate; a virtual machine (VM), e.g., Maté [7] and JVM [8], can naturally support loading and executing a very compact code, but requires additional runtime support. Moreover, the VM code is less expressive than the native code.

Contrary to the abovementioned mechanisms, dynamic loading is an efficient way to enable sensor nodes to be reprogrammable: on one hand, it allows disseminating

* Corresponding author.
E-mail addresses: dongw@zju.edu.cn (W. Dong), chenc@zju.edu.cn (C. Chen), bjj@zju.edu.cn (J. Bu), huangchaohun@foxmail.com (C. Huang).

1570-8705/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.adhoc.2012.07.007

the loadable module, which is much smaller than the entire application image required by the bootloader mechanism; on the other hand, it allows executing the native code, which is much more efficient for execution than the VM instructions required by the virtual machine mechanism.

There also exist a number of differential-based approaches to reduce the transferred code size [9–11]. Existing differential-based approaches require the knowledge of exact software configurations on the sensor nodes. If sensor nodes are running different versions of their software (which is typical when a fraction of nodes are being updated for testing), differential-based approaches do not scale [8].

In this paper, we explore the dynamic modular design for micro embedded systems, such as sensor nodes. Our design and implementation are guided by the following requirements, including (1) small module size, (2) flexible programming model, and (3) fine-grained configuration.

There is a tradeoff between the reprogramming efficiency and reliability. The use of additional reprogramming techniques, such as general compression and differential encoding, can further reduce the transferred file size. However, they require a more complicated design on the sensor nodes. Complicated mechanisms are likely to be causes of failures [12,13]. Our design focuses on reducing the unnecessary fields in the module file format while retaining its essential functionalities. Dynamic linking and loading of the modules are sophisticated techniques on PCs and we believe that they can also be reliably and efficiently performed on the sensor nodes.

While some existing works address one or two requirements, few of them present a holistic solution.

The existing modular approach, CELF [8] (based on Contiki OS [14]), only redefines long data types to short ones for low-end microcontrollers. This approach is not enough for aggressively optimizing the module size because the metadata (such as the relocation table and the symbol table) still occupies a large fraction of the module size.

MELF (based on SOS [15]), being small in module size, has several limitations. First, it needs a special SOS module header to be defined by programmers. Second, it does not support defining global variables in a module. Third, it does not support functions calls by names for inter-module communications. These limitations should be avoided.

Therefore, in this paper, we propose a new modular approach, SELF, to have a good tradeoff between the module size and the module's functionalities (e.g. for enabling flexible programming, fine grained configuration). At the high level, we maintain ELF's basic structure, e.g. the data, bss, text, sym, reloc sections while reducing unnecessary overheads for resource-constrained sensor nodes.

We have implemented the modular approach based on a micro embedded OS, SenSpire OS [16], for AVR and MSP430 platforms. The evaluation results show that the proposed SELF module file format is 4.6–7.6 times smaller than the standard ELF format, and is 1.6–2.4 times smaller than the CELF format. SELF retains necessary information to enable flexible modular programming and inter-module communications. We have further developed a long-term energy efficiency model to explore the tradeoffs of different reprogramming approaches.

The contributions of this paper are highlighted as follows:

- We design, implement, and evaluate the SELF format which has a good tradeoff in the module size and module's functionalities. We quantitatively compare our methods with state-of-the-arts, including MELF [15], CELF [8], and Minilink [17].
- We develop a long-term energy model considering both reprogramming energy consumption and execution energy consumption. Based on this model, we discuss tradeoffs of different reprogramming approaches. We further study the impact of several key system parameters to the network lifetime.

The rest of this paper is structured as follows: Section 2 presents our design in detail. Section 3 shows the evaluation results. Section 4 describes related work. Finally, Section 5 concludes this paper.

2. Design

This section describes our design approach in detail. Section 2.1 gives the design overview. Section 2.2 describes techniques we employed to reduce the module file size. We quantify sizes of different components constituting the entire module, and compare our approach to two well-known module formats for sensor systems (i.e., CELF [8] and MELF [15]). Section 2.3 describes the dynamic loading process which carefully handles module dependencies, thus enabling a fine-grained module configuration. Section 2.4 describes three inter-module communication methods and discusses their tradeoffs.

2.1. Overview

Fig. 1 depicts the architectural overview of our modular design approach based on SenSpire OS. The module code is first compiled by the GCC utility into a standard ELF, which is a common file format for linking and loading on traditional PCs [18]. We employ module reduction techniques

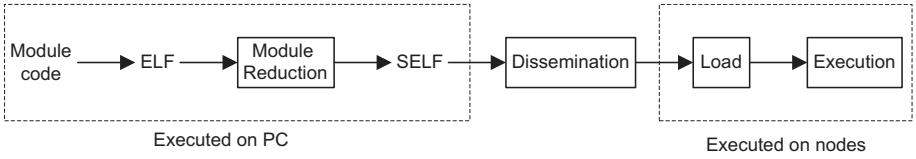


Fig. 1. Overview of our modular design approach.

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

to reduce the module file size, which is important for efficient dissemination. We have developed a tool, called elftoself.exe, for reducing the module file size (described in Section 2.2). The resulting file format is called SELF. The SELF module is then disseminated to all nodes in the network by the dissemination protocol, e.g., Deluge [6]. The dissemination protocol saves the received module onto the external flash of each node. After receiving an entire module, the SELF loader starts loading the module for execution. The SELF loader loads the module in a fine-grained manner, handling module dependencies (described in Section 2.3). After the loading process, the code in the module can be executed. SELF supports three methods of inter-module interactions and communications (described in Section 2.4).

2.2. Module size reduction

The module file size greatly impacts the energy efficiency during code dissemination. Therefore, we need to reduce the module file size at the PC side before it is injected into the network. This section describes techniques we employed to reduce the module size.

2.2.1. Basic reduction

We redefine the basic data types in standard ELF to the shortened ones in SELF, as summarized in Table 1. This approach causes data structures in ELF to be shortened, hence reduces the module file size. Table 2 compares the sizes of the main data structures in ELF [18], CELF [8], MELF [15], and SELF [19].

Note that some data structures in CELF are not exactly half of the sizes in ELF. This is because some one byte fields cannot be reduced down further. Some data structures in SELF are smaller than that in CELF because we further reduce the sizes by eliminating some unused data fields.

2.2.2. Code reduction

Code constitutes a large fraction of the module size. Hence optimization of the code size is important for reducing the module size. GCC [20] provides many options to optimize the code. We use the "-Os" option to optimize for the code size. Another design choice is whether to use position independent code (PIC) or relocatable code. It is discussed in [8] that PIC has several limitations, e.g., a maximum of 4 KB relative jumps, lack of architecture or compiler support (e.g., MSP430-GCC does not support PIC). SOS 1.3 uses a GCC option "-mshort-calls" to replace all direct branch instructions (e.g., jmp/call) to relative ones (e.g., rjmp/rcall). Hence, it suffers from the abovementioned limitations for MSP430 platforms. SOS 2.0 and Contiki OS choose to use relocatable code. This is a more portable design choice. However, it incurs an increase in the module file size for AVR platforms. We address this issue by two techniques. First, we encapsulate different GCC options for different platforms. For example, for MSP430 platforms, we use relocatable code completely. For AVR platforms, we use PIC whenever it is possible. Second, to overcome the 4 KB limit of relative branch instructions for AVR platforms, we use the GCC option "-Wl, -relax", which can automatically replace direct branch instructions that are within 4 KB to relative ones while leaving direct branch instructions with larger jumps unchanged. We do relocation in the loader for both platforms. Fig. 2 gives graphical illustration of the code reduction techniques. We can see that this technique optimizes the module size for AVR platforms in two aspects. First, jmp/call require 4 bytes while rjmp/rcall require 2 bytes. Second, the number of relocation entries can be reduced because relocation is not needed for relative branch instructions.

Most platforms support the use for relocatable code, and can safely use the template for MSP430 platforms. For platforms that additionally support the use of PIC, we can use additional flags (e.g., -mshort-calls, -Wl, -relax) to allow the use of PIC since this will further reduce the code size.

Table 1
Data types redefinitions.

Name	Definition in ELF/bytes	Definition in SELF/bytes
Elf_addr	Unsigned int/4	Uint16_t/2
Elf_half	Unsigned short/2	Uint8_t/1
Elf_off	Unsigned int/4	Uint16_t/2
Elf_sword	Int/4	Int16_t/2
Elf_word	Unsigned int/4	Uint16_t/2

Table 2
Sizes of main data structures in ELF, CELF, MELF, and SELF.

Data structure	ELF	CELF	MELF	SELF
ELF header	38	20	4	8
ELF section header	40	20	8	8
Symbol entry	16	9	4	4
Relocation entry	12	6	8	6

2.2.3. Relocation table reduction

The relocation table contains relocation entries for symbol relocation. In the standard ELF, each relocation entry contains an index to the symbol and a pointer to the unresolved reference. The GCC compiler generates a relocation entry for each unresolved reference. To reduce the number of relocation entries, we employ the chained references technique [18]. The basic idea of this technique is trying to merge the relocation entries for the same unresolved symbol, because references to the same symbol in different locations of the program must point to the same address. To achieve this goal, we use references to the same unresolved symbol in the program (which contain useless addresses before relocation) to create a linked list (as illustrated in Fig. 3). Therefore, in SELF, each entry in the relocation table requires two values: an index to the symbol and a pointer to the first reference of that symbol (i.e., the header of the linked list). Our loader will do relocation by traversing the linked list. By this technique, in SELF, the relocation table grows with the number of unique symbols, instead of the number of references.

It is worth noting that the current standard mechanism does not use this technique. This is because mote software is expected to be simple and there is even no loader implementation on typical sensor applications. For module files with chained reference, the linking and loading process

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

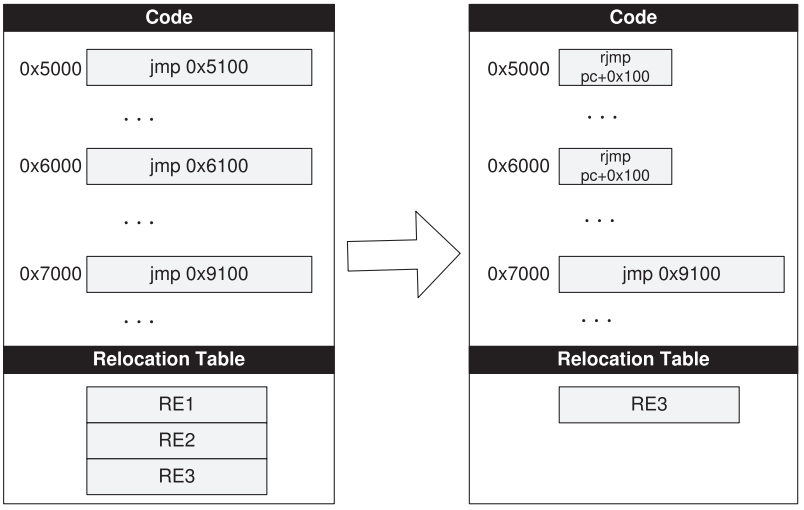


Fig. 2. Code reduction techniques.

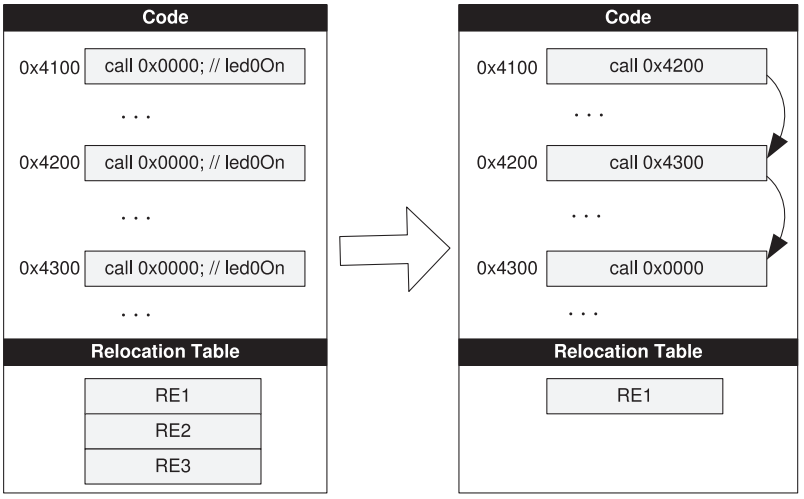


Fig. 3. Chained reference technique.

will be more complicated since the loader needs to traverse the linked list to perform relocation. We believe that this overhead is worthwhile as it can significantly reduce the module size for efficient dissemination (radio consumes much more energy than the CPU).

2.2.4. Symbol table reduction

The symbol table contains symbol entries of two categories: The first category includes symbols for compile-time linking and the second category includes symbols for run-time linking. We reduce the number of symbol entries in two ways. First, we eliminate symbols of the first category by parsing the ELF and removing unreferenced symbols. Second, we eliminate the system call symbols by allocating a system call jump table and pre-linking to the corresponding slot. The existence of the system call jump table allows kernel differences on different nodes. The kernel function implementing a system call can be located at different addresses as long as the system call jump table slot points to the correct address. We have noticed that the CELF module does not pre-link to system calls, hence incur a larger overhead in the symbol table. Moreover, the existence of a system call symbol also requires its string representation in the string table. As system call function names can be arbitrarily long, they constitute a large fraction of the module size. By pre-linking to system calls, we are able to reduce the symbol table size and the string table size.

2.2.5. SELF module format summary

The resulting SELF format includes the following components (as shown in Fig. 4).

- The SELF file header. The size is denoted as $S(hr)$.
- The SELF section headers. The total size is expressed as $S(shr) = s(shr) \cdot N(shr)$, where $s(shr)$ is the size of a single section header and $N(shr)$ is the number of section headers.

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

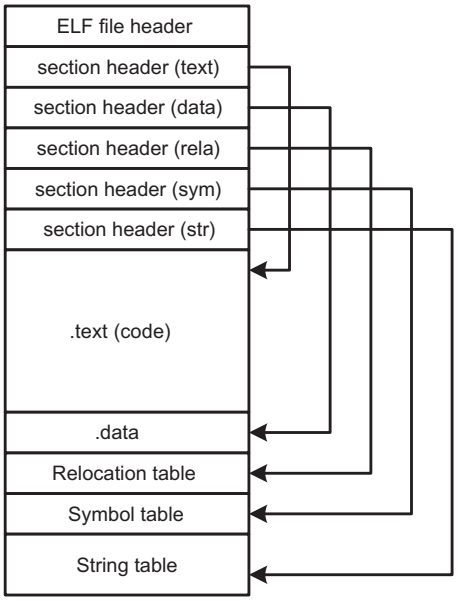


Fig. 4. The SELF file format.

- The code (i.e., the .text section). The size is denoted as $S(text)$.
- The uninitialized data (i.e., the .bss section). Note that the values of uninitialized data are known to be zero. Hence it does not occupy space in the module file. However, it does occupy space in RAM. We denote its RAM size as $S(bss)$.
- The initialized data (i.e., the .data section). The size is denoted as $S(data)$.
- The relocation tables, including the relocation table for the code (i.e., the .rela.text section), and the relocation table for the initialized data (i.e., the .rela.data section). The size is expressed as $S(rela) = s(rela) \cdot N(rela)$, where $s(rela)$ is the size of a single relocation entry and $N(rela)$ is the total number of relocation entries.
- The symbol table. The size is expressed as $S(sym) = s(sym) \cdot N(sym)$, where $s(sym)$ is the size of a single symbol entry and $N(sym)$ is the number of symbol entries.
- The string table. The size is expressed as $S(str) = \sum_{nm \in \mathbb{E}} sizeof(nm)$, where $\mathbb{E}$ is the set of exported names in the program. Note that $sizeof(\cdot)$ also accounts for the ending "\0".

Hence, the total module size $S(m)$ can be expressed as,

$$S(m) = S(hr) + s(shr) \cdot N(shr) + S(text) + S(data) + s(rela) \cdot N(rela) + s(sym) \cdot N(sym) + \sum_{nm \in \mathbb{E}} sizeof(nm) \quad (1)$$

The sizes for $S(hr)$, $s(shr)$, $s(rela)$, $s(sym)$ are actually given in Table 2 for different module file formats. Given the quantitative size of each component, we are able to compare SELF with CELF and MELF in several important aspects.

- First, CELF only redefines long data types to short ones to reduce the module size without a deep look into the file contents. SELF further eliminate unused data fields in some structures, and more importantly, employ specific techniques to reduce the size of code, relocation tables, and the symbol table, which constitute a large fraction of the module size. For example, the number of relocation entries, $N(rela)$, for CELF (and also MELF) is the number of total number of references while $N(rela)$ is the number of unique references in SELF; The number of symbols, $N(sym)$, for CELF includes system calls which may be invoked frequently by the applications, while $N(sym)$ for SELF does not include this overhead.
- Second, MELF does not include the relocation table for data, and it does not include the string table. Generally speaking, MELF has a very small module size. The process of defining global variables is more demanding in comparison to SELF. As we will demonstrate in Section 3.2, MELF requires global variables to be declared in a pre-defined data structure, and get initialized in the module handler function [15]. Direct communications between MELF modules can only be via function IDs, rather than function names.

2.3. Loading and unloading

After receiving an entire module (by a dissemination protocol), it is saved onto the external flash. The SELF loader starts loading and unloading modules when it is triggered by a command issued from the sink. It automatically handles module dependencies. For example, it automatically loads modules involved in the dependency chain when necessary, and unloads modules when they are not needed. This section describes the loading and unloading process performed by the SELF loader.

2.3.1. Loading

The loading process is responsible for loading the module from the external flash to RAM (for data) and the program flash (for code). This process is described as follows.

- Load. The loader reads the SELF file header. It checks for platform compatibility and file correctness. It also checks for the existence of all dependent modules. The loader starts loading the current module only if the current module is not already loaded into memory and all the dependent modules are already loaded or available on the external flash.
- Alloc. The loader reads section headers to obtain detailed information about the corresponding sections. According to the obtained sizes of .data, .bss, and .text sections, the loader allocates memory spaces as follows. It allocates $S(RAM) = S(data) + S(bss)$ bytes in RAM for .data and .bss sections, and it allocates $S(flash) = \sum_{sym,nm \in \mathbb{E}}(s(sym) + sizeof(nm)) + S(text)$ bytes in program flash. Note that additional space for the exported symbols and symbol names must be allocated so that other modules can invoke these exported functions. The loader aborts if there is not enough memory.

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

- List. The loader keeps a loading list. At this step, the loader adds the current module to the loading list.
- The loader Loads all modules that the current module depends on. This is a recursive process. If the loader finds that the reference counter of the dependent module is larger than 0, the loader skips loading this module and proceeds because it is already loaded. When the dependent module is already in the loading list, there must be circular dependencies, hence the loader aborts. When all dependent modules complete the loading process, the loader goes to the next step.
- Copy. (i) The loader first copies the .text section from the external flash to RAM segment by segment, performing relocating according to the .rela.text section. The loader next copies the relocated code to the program flash. (ii) The loader also writes exported symbols and symbols names to the program flash just before the code section. It relocates the symbol entries to reflect their actual addresses for run-time name resolving. (iii) The loader loads the .data section to RAM, performing relocating according to the .rela.data section. (iv) The loader initializes the .bss section to zero.
- Exec. The loader removes the current module from the loading list, increments the module's reference counter by 1, and finally, passes an init message to the module.

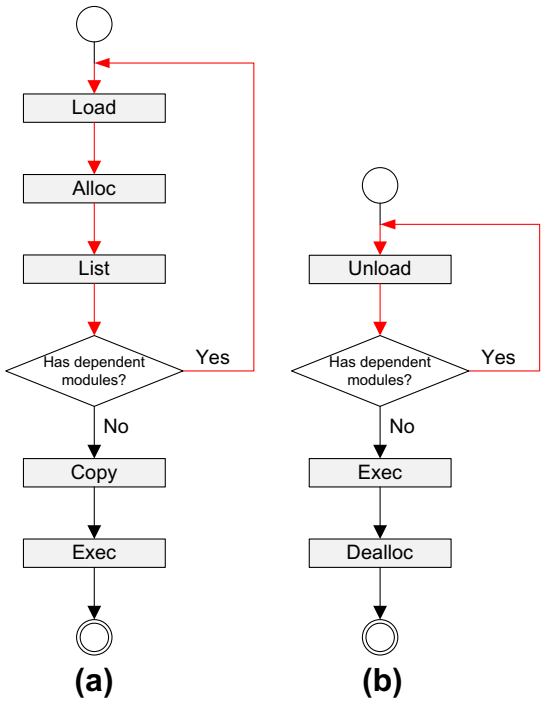


Fig. 5. Loading and unloading. (a) Loading. (b) Unloading.

2.3.2. Unloading

The unloading process is described as follows:

- Unload. The loader decrements the reference counter of the current module by 1. The loader goes to the next step only when the reference counter becomes zero.
- The loader Unloads all dependent modules before the current module. This is a recursive process.
- Exec. The loader passes a finish message to the module, allowing the current module to relinquish system resources allocated by the program.
- Dealloc. The loader deallocates the memory space occupied by the current module.

2.3.3. Summary

Fig. 5 gives the workflow of the loading and unloading process in the loader. The loading and unloading process described above has the following features:

- *Reliability*. The SELF loader loads a module only when all dependent modules are loaded into the memory. The SELF loader also avoids circular dependencies by keeping a loading list.
- *Low configuration*. The SELF loader automatically unloads unneeded modules whose references counters equal to zero.

The implementation of the SELF loader is lightweight on current sensor nodes. For instance, for the msp430 platform, the ROM overhead is 2948 bytes and the RAM overhead is 44 bytes. This overhead is acceptable for current sensor nodes.

2.4. Module communications

SELF modules communicate via three mechanisms which will be described in the following subsections respectively.

2.4.1. Calls with one indirection

As described in Section 2.3.1, a module saves information of its exported functions in the program flash, for run-time function looking up. SELF loader supports two kinds of inter-module calls (as illustrated in Fig. 6). The first approach is via function IDs. A module can dynamically register its function address by a system call `sys_register_fn`. After registration, another module can obtain the registered function address by calling another system call `sys_get_fn`. After this, the module can invoke the registered function by dereferencing the obtained function pointer. The second approach is via function names. A module can specify its exported functions in the program code. Information about all the statically exported functions is stored in the program flash. Therefore, the module does not need to explicitly register for these functions. Another module can obtain the address of an exported function by a similar system call `sys_get_fnex`. After this, the module can invoke the exported function by dereferencing the obtained function pointer.

The first approach requires inter-module communications via function IDs, which is less natural than that by function names. However, this approach does not require the corresponding symbol entry and its string representation encoded in the module file, which reduces the module size. The second approach is more natural for application

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

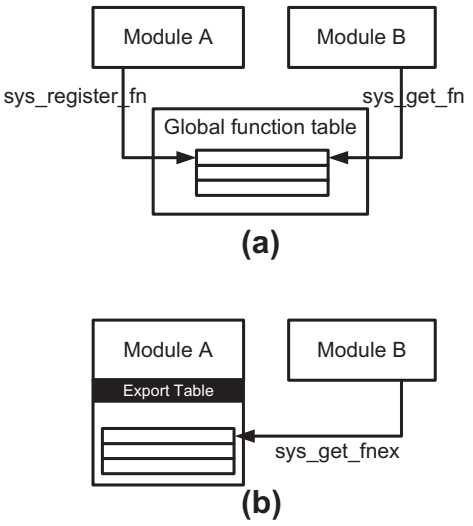


Fig. 6. Calls with one level of indirection. (a) calls via function IDs and (b) calls via function names.

programmers, but requires extra information to be encoded into the module file.

Also note that we always use one level of indirection. Direct function calls are not allowed for inter-module communications in order to decouple module dependencies. If we allow direct function calls, then when a module is updated, we need to update all modules that depends on this module. This is because the references in these modules will be incorrect when the dependent module is updated. This causes extra implementation complexity, more flash writes, and does not scale when more modules are involved.

2.4.2. Message passing

As a special kind of inter-module calls with one indirection, SELF module supports passing synchronous messages to a default module handler function. The module handler function is also responsible for receiving the `init` message and `finish` message when the module is loaded or unloaded (Section 2.3). When a module handler function receives a message, it uses a "`switch...case`" statement to handle different messages. This approach does not require explicit registration or encoding extra information into the module file, but it is slightly slower than approaches described in the previous subsection because extra overhead is involved in order to differentiate different messages.

2.4.3. System calls

Some kernel functions will be accessed frequently by applications. We implement a lightweight system call mechanism. A system call jump table is located at a fixed address in the program flash. All system calls in the application program are pre-linked to its corresponding address in the jump table slot. The existence of the system call jump table allows a loose coupling between application modules and kernel modules. When a kernel module is updated, it only needs to update the corresponding system call jump table slots. Compared to approaches described in the previous two subsections, the system call mechanism allows calling to the functions by names. It does not encode information into the module file. However, it requires the system call jump table to be reserved in the program flash.

We currently use system jump table for storing system functions provided by the kernel. For calls within a module, we perform relocating prior to execution since this will further reduce the indirection cost. For calls outside a module, we provide mechanisms similar to system calls. To make a function accessible from other modules, we need to first register the function, and then make calls to registered function.

3. Evaluations

In this section, we evaluate our modular design approach. Section 3.1 compares SELF with standard ELF and also shows the overall statistics of SELF. Section 3.2 compares SELF with MELF (for SOS) from three perspectives. Section 3.3 compares SELF with CELF (for Contiki OS). Section 3.4 studies the reprogramming energy efficiency. Finally, Section 3.5 presents a long-term energy model to study our work in a broader spectrum of network reprogramming approaches.

We conduct our experiments based on three different OSes, i.e., SenSpire OS, SOS, and Contiki OS. (1) We implement ELF, SELF, and CELF based on SenSpire OS. Hence, we are able to compare ELF, SELF, and CELF based on SenSpire OS using 11 SenSpire OS benchmarks. (2) We did not implement the MELF format based on SenSpire OS because MELF requires some OS specific features, e.g., message passing, module header processing. In addition, the programming style of SenSpire OS applications is also different from SOS (which uses message handlers). (3) In order to compare SELF, MELF, and CELF more faithfully, we implement SELF based on SOS 2.0 and Contiki OS 2.4, respectively. In Section 3.2, we compare SELF with MELF based on SOS 2.0 using five SOS benchmarks. In Section 3.3, we compare SELF with CELF based on Contiki OS 2.4 using five Contiki OS benchmarks.

3.1. Module size

We examine the SELF module size based on 11 SenSpire OS benchmarks for MSP430 platforms. For AVR platforms, similar results can be obtained [19].

Fig. 7 shows the breakdown of SELF module size. The total sizes of these modules vary from 170 bytes to 918 - bytes, depending on the complexity of the benchmark. The fractions of metadata vary from 0.32 to 0.51.

It can be seen from Fig. 7 that there seems a correlation between task complexity and the fraction of metadata in the module. To investigate such correlation, we use a scatter plot to show the relationship. We use the SELF size to measure the complexity of a benchmark. Fig. 8 shows the results. We can see that benchmarks with larger SELF size have smaller fractions of metadata. To further quantify the correlation, we calculate the Pearson product-moment correlation coefficient between the two parameters (i.e. SELF

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

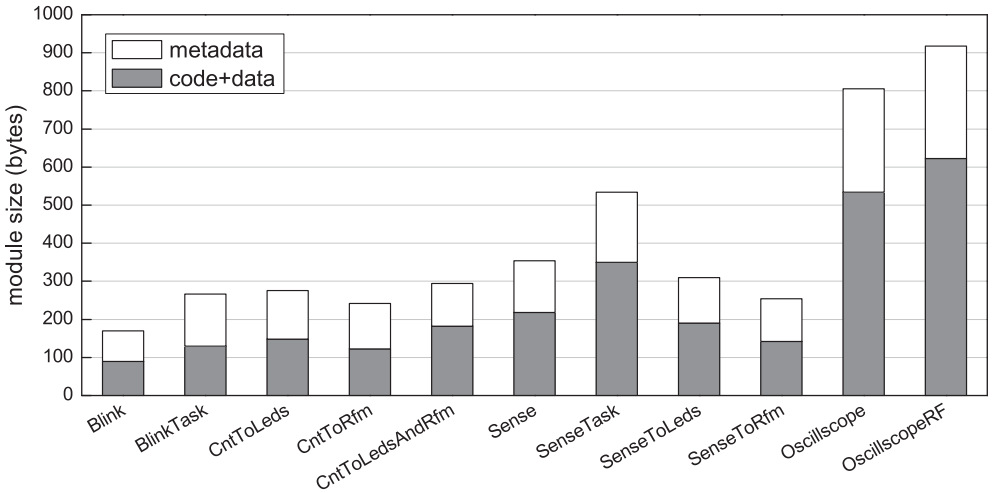


Fig. 7. SELF module sizes of 11 SenSpire OS benchmarks.

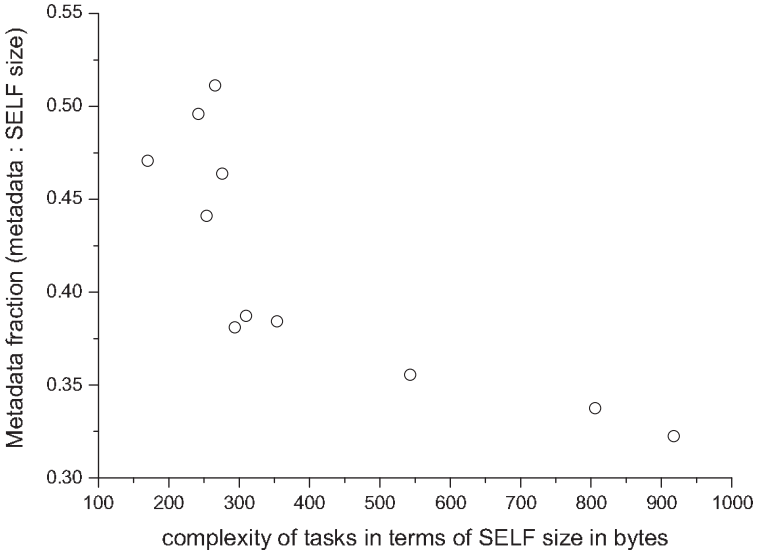


Fig. 8. SELF size vs. metadata fraction (for 11 SenSpire OS benchmarks).

size and fraction of metadata) according to the following formula,

$$r = \frac{\sum_{i=1}^{n}(S_i - \overline{S})(M_i - \overline{M})}{\sqrt{\sum_{i=1}^{n}(S_i - \overline{S})^2}\sqrt{\sum_{i=1}^{n}(M_i - \overline{M})^2}} \quad (2)$$

where i denotes the ith benchmark, $n = 11$ is the total number of benchmarks examined, S_i is the SELF size of the i-th benchmark, M_i is the fraction of metadata in the ith benchmark, $\overline{S}$ is the mean value of $\{S_i\}_{i=1}^{11}$, and $\overline{M}$ is the mean value of $\{M_i\}_{i=1}^{11}$. We find out that $r = -0.8$, indicating that there is a strong negative correlation between task complexity and the fraction of metadata, i.e. when the code becomes more complex, the fraction of metadata becomes less significant.

We only reveal the statistical relations between the task size and the metadata under investigation with no intention to reveal the causal relationship. To understand the underlying reasons, it is worth noting that the metadata consists of module header, relocation table, and other parts. While the size of some parts (e.g. relocation table) is proportional with the task complexity, the size of some other parts (e.g. module header) is fixed. The fraction of metadata can be calculated as $f = (F + P)/C$ where F denotes the fixed part, P denotes the proportional part, and C denotes the code size. We investigate all the benchmarks and find that the proportional part occupies roughly the same fraction. The fixed part will be insignificant with a large code size. Therefore, the fraction of metadata becomes less significant when the code becomes more complex. It is worth noting that for our benchmarks, we

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

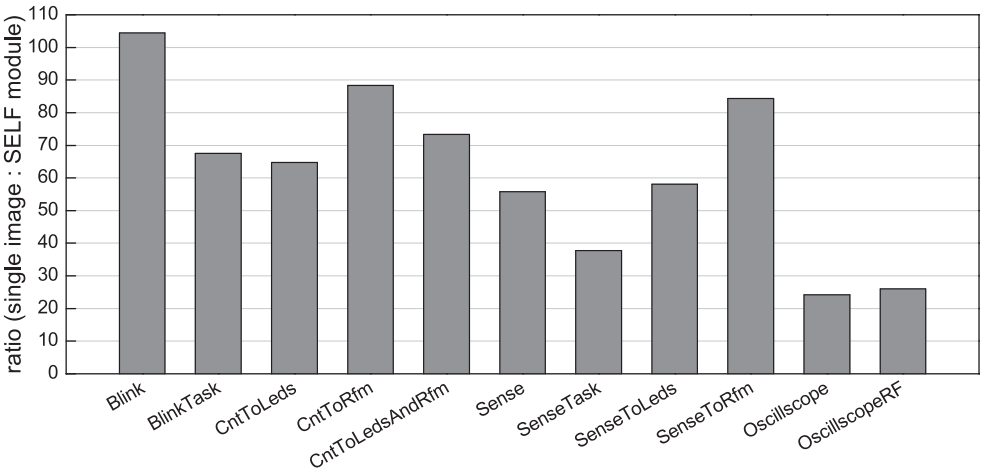


Fig. 9. Comparison of SELF module size and full image size (for 11 SenSpire OS benchmarks).

observe that as code complexity increases, the fraction of metadata becomes less significant. This observation may not be true for other benchmarks.

Fig. 9 shows the ratios of the full image size divided by the SELF module size. We can see that our modular approach can reduce the transferred data size significantly: the full image size is 26–104 times larger than the SELF module size.

We again examine the correlation between task complexity (which is defined to be the SELF size) and relative reduction of SELF over full image. Fig. 10 shows the scatter plot of SELF size and the ratio. The calculated Pearson product-moment correlation coefficient is −0.89, indicating a strong negative correlation between task complexity and the ratio. This implies that the improvement of SELF is more significant for simpler applications.

Fig. 11 shows the ratios of standard ELF size divided by the SELF size. We can see that the standard ELF size is 4.6–7.6 times larger than SELF. This illustrates that the module format for micro embedded systems should be tailored in order to improve the reprogramming performance during code dissemination.

Fig. 12 shows the effects of two most important techniques in reducing the module size. (1) The basic reduction combined with code optimization (Sections 2.2.1 and 2.2.2) reduces the size by more than 50%. (2) The use of system call gate further reduces the code size to the one approaching the final SELF size. This is because the use of system call gate effectively reduces both the symbol table size, relocation table size, and the string table size.

Dissemination time and transmission overhead are important metrics for code dissemination. We perform testbed experiments with 24 telosB notes in a 4×6 grid. The internode spacing is approximately 35 cm and the power level is configured to 1 in order to simulate the multihop performance. Figs. 13 and 14 show the dissemination

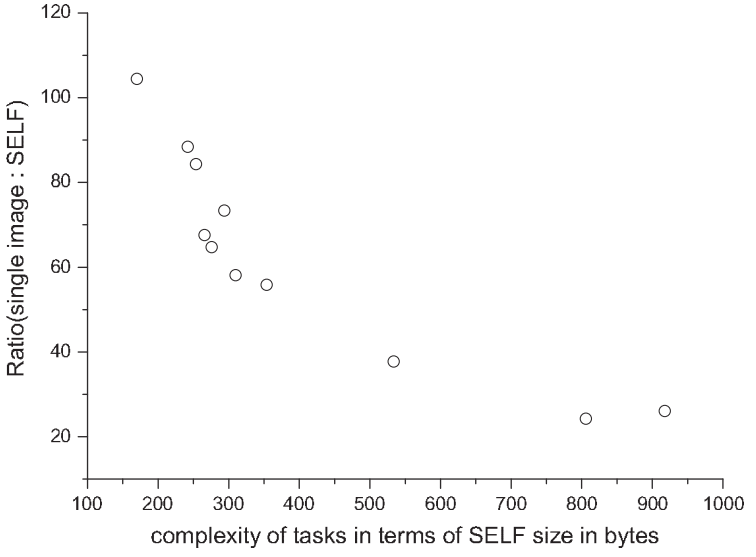


Fig. 10. SELF size vs. the ratio (singe image/ SELF, for 11 SenSpire OS benchmarks).

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

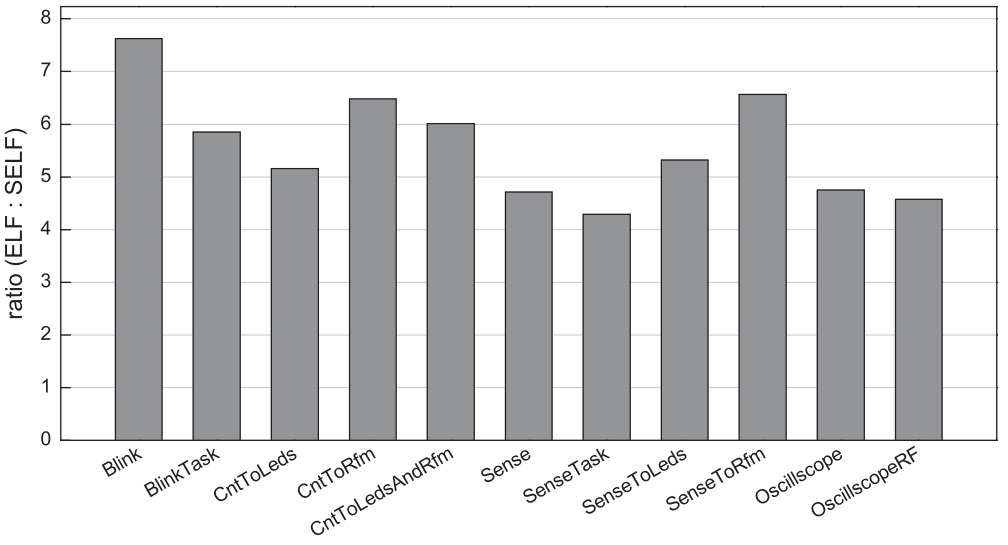


Fig. 11. Comparison of SELF size and standard ELF size based on SenSpire OS using 11 SenSpire OS benchmarks.

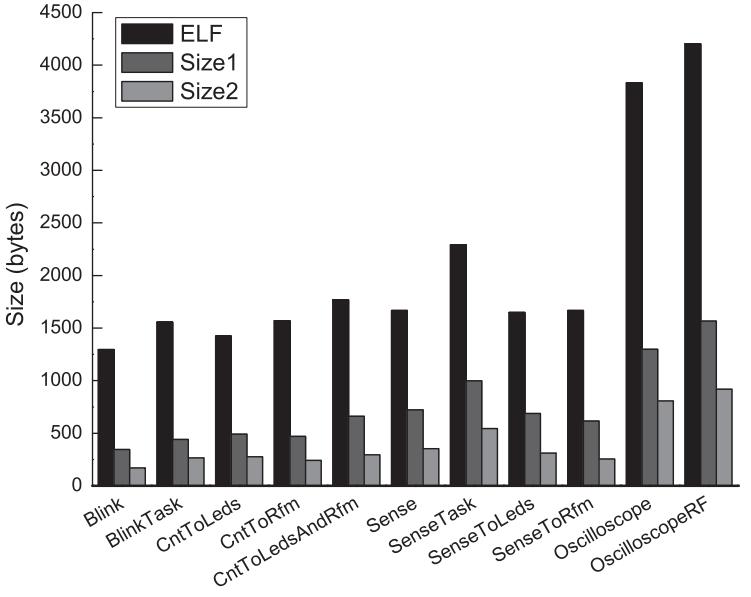


Fig. 12. Effects of two most important techniques. Size1 denotes the module size after applying techniques described in Sections 2.2.1 and 2.2.2. Size2 denotes the module size after applying the technique of system call gate.

time and transmitted data packets. We can see that the module size proportionally impacts the transmission overhead and completion time.

3.2. Comparison with MELF

Module size. We implement SELF based on SOS 2.0.1 in order to compare the module size of SELF with that of MELF. Table 3 shows the module file sizes based on five SOS benchmarks for the AVR platform. The MELF size is slightly larger than that of SELF.

It is worth noting that SELF adopts several techniques to reduce the module size. First, SELF uses PIC for the AVR platform whenever possible. Second, SELF employs the chained reference technique to reduce the relocation table size. With the same `surge` example, the chained reference technique can reduce the number of relocation entries from 29 to 18. Third, the string table in SELF only contains external function names.

Programming. SELF enables a more convenient programming model. For example, SELF allows directly defining global variables and performs data relocation.

The following code snippet shows how to define a global variable, seqno, to a SOS module. First, we need to add the variable to a module state structure named surge_state_t. Second, we need to allocate memory space for

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

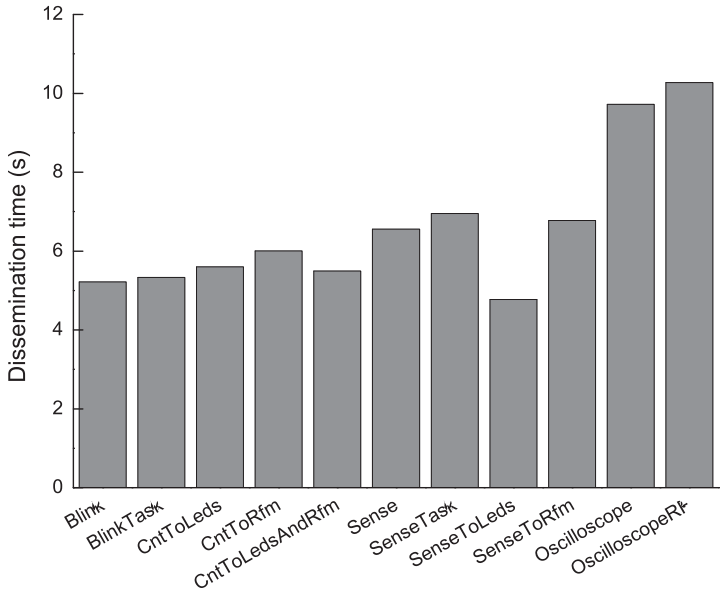


Fig. 13. Dissemination time (s).

the module state structure in the SOS module header. Third, we need to initialize the variable in the MSG_INIT message handler.

```
typedef struct {
  func_cb_ptr get_hdr_size;

  int16_t timer_ticks;
  uint32_t seq_no;
  sos_pid_t dest_pid;
  SurgeMsg* smsg;
} surge_state_t;
...
static const mod_header_t mod_header
  SOS_MODULE_HEADER = {
    .mod_id = SURGE_MOD_PID,
    .state_size = sizeof (surge_state_t),
    .num_sub_func = 1,
    .num_prov_func = 0,
    .platform_type = HW_TYPE /* or PLATFORM_ANY
*/,
    .processor_type = MCU_TYPE,
    .code_id = ehtons (SURGE_MOD_PID),
    .module_handler = surge_module_handler,
  };
  ...
  case MSG_INIT:

  {
    s->seq_no = 0;
  break;
  }
```

This complexity mainly comes from the fact that MELF does not support relocation in the data section. With SELF, the programming efforts can be greatly reduced since it directly supports defining global variables (as shown in the following code snippet).

```
class Surge {
  uint32_t seqno = 0;
  void start () {}
  ...
}
```

Module dependency. Another difference from both MELF and CELF is that we handle module dependencies. This allows modules to be developed in a fine-grained manner. Also, this improves system reliability while at the same time keeping system configuration to its minimum requirement.

3.3. Comparison with CELF and Minilink

Compared to CELF, we reduce another major contributor to the module size, i.e., the system calls. CELF uses a runtime linking technique to relocate these references. This raises two problems. First, the corresponding symbol entries and string names (which can be arbitrarily long) must be encoded in the module file, which increases the module size. Second, when the kernel gets updated all application modules need to be re-linked and re-loaded. Actually, Contiki OS does not address the second issue altogether: it does not allow dynamically updating the OS kernel.

Fig. 15 shows the ratios of the CELF size divided by the SELF size based on 11 SenSpire OS benchmarks. We can see that for a wide range of applications, CELF is 1.6–2.4 times larger than SELF.

In order to see SELF's generality, we implement SELF based on Contiki OS 2.4 and compare the module size of

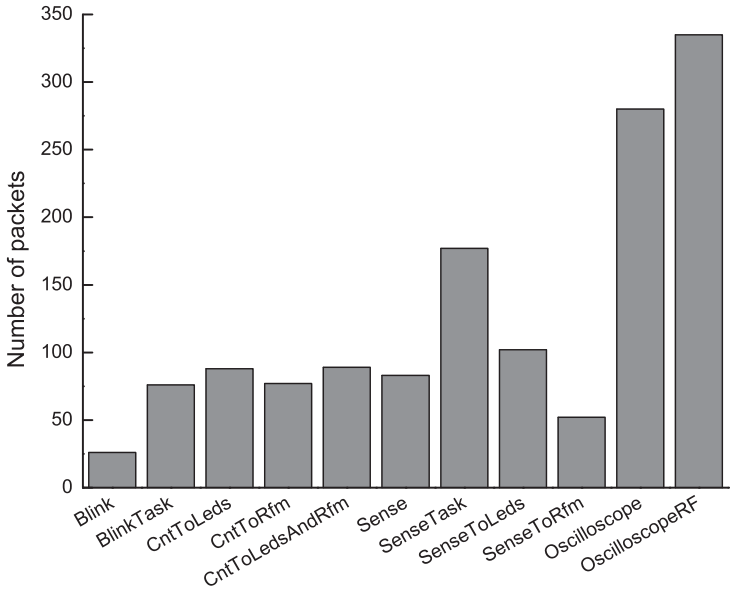


Fig. 14. # Of transmitted data packets.

Table 3
Module size comparison of MELF and SELF based on SOS (bytes).

Benchmark	MELF	SELF
Surge	754	679
Pingpong	552	530
Neighbor	1740	1564
Tree_routing	1366	1214
AODV	6728	5730

CELF and SELF based on five Contiki benchmarks for the MSP platform. Table 4 shows that CELF is 1.3–1.6 times larger than SELF.

Table 4 also shows the results for Minilink [17]. Minilink [17] is a recent work that enables stateful mobile modules for sensor network. Minilink incorporates several techniques to reduce the module file size, including eliminating common prefix in the symbol table, placing the relocation entries in the code section, etc. Compared to SELF, Minilink enables stateful migration of code. This is achieved at additional cost of serialization of relevant data. On the other hand, SELF has included some other reduction techniques, including the chained reference technique, the use of system call gate. In addition, SELF enables fine-grained module configuration. We have also proposed a network lifetime model for better understanding of various reprogramming approaches.

There are several recent sensor OSes architected with a modular design. For example, LiteOS [21] also supports dynamic loadable modules. It utilizes a differential patching idea with a mathematical model. However, it requires a training process at the PC side to ensure the correctness of the model. It remains challenging how many training data should be sufficient to ensure the system reliability.

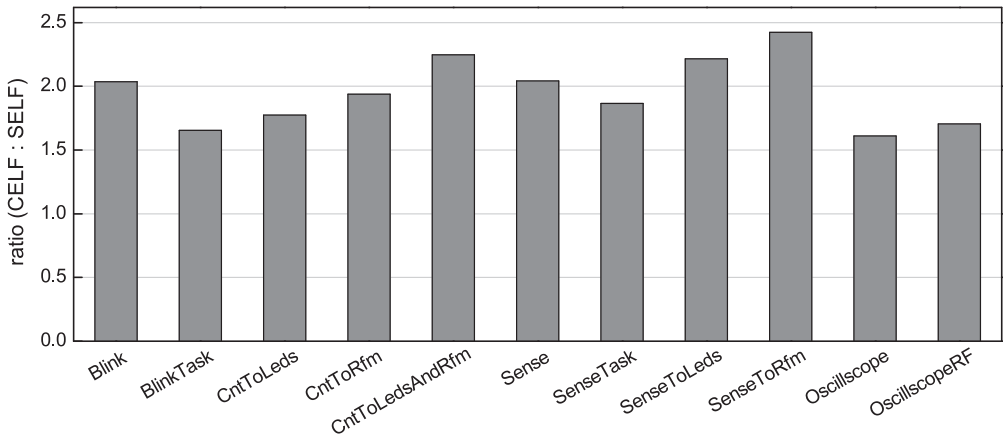


Fig. 15. Comparison of CELF size and SELF size based on SenSpire OS.

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

Table 4
Module size comparison among CELF, Minilink, and SELF based on Contiki OS (bytes).

Benchmark	CELF	Minilink	SELF
Hello-world	606	541	390
Radio-test	2139	1740	1620
Test-button	949	806	661
Test-cfs	1679	1474	1273
Example-coffee	1963	1680	1540

3.4. Reprogramming energy consumption

We use E_{reprog} to denote the energy consumption during reprogramming. According to [8],

$$E_{reprog} = E_p + E_s + E_l + E_f \tag{3}$$

where E_p is the energy consumption during dissemination; E_s is energy consumption to store the object onto the external flash. E_l is the energy consumption of dynamic relocation; E_f is the energy consumption to write the relocated code to the program flash.

$$E_p = v \cdot I_{radio} \cdot c_0 \cdot S(m) \tag{4}$$

where $S(m)$ is the transferred data size, $c_0 \cdot S(m)$ models the overall dissemination time, which can be approximately expressed as a linear function of $S(m)$. We assume the radio is always on during code dissemination. For the current Deluge protocol [6], during bulk data dissemination, the radio should be turned on. This is because the Deluge protocol is not designed to be work on LPL. The current collection protocol, CTP, can work on LPL MAC. Whether an application can work with low power MAC depends on both the implementations of the application and MAC.

The constant factor, c_0, depends on a number of factors such as link loss rates, and network scale. For example, in our previous study [22], we have observed that for a 10×10 square topology, c_0 varies from 100 s/KB to 700 s/KB with inter-node spacing varies from 5 feet to 20 feet. Without loss of generality, we assume c_0 = 200 s/KB here.

$$E_s = e_s \cdot S(m) \tag{5}$$

where e_s is the average energy consumption for storing one byte to the external flash.

$$E_l = e_l \cdot N(rela) \tag{6}$$

where $N(rela)$ is the number of relocation entries, and e_l is the average energy consumption to perform one relocation.

Table 5
Energy consumption of different modular approaches (mAs).

Benchmarks	ELF	CELF	SELF
Blink	15,557	4153	2040
BlinkTask	18,679	5282	3193
CntToLeds	17,094	5882	3313
CntToRfm	18,823	5630	2905
CntToLedsAndRfm	21,224	7935	3529
Sense	20,023	8679	4249
SenseTask	27,514	11,968	6410
SenseToLeds	19,783	8247	3721
SenseToRfm	20,023	7394	3049
Oscillscope	46,001	15,581	9675
OscillscopeRF	50,443	18,799	11,020

$$E_f = e_f \cdot S(flash) \tag{7}$$

where $S(flash)$ is the size of the program code that is written onto the program flash, and e_f is the average energy consumption for writing one byte to the program flash.

We use the above model to analyze the energy consumption of different modular approaches. Table 5 shows the results. We can see that SELF has the least energy consumption during reprogramming.

3.5. A long-term energy efficiency model

In this section, we continue to analyze the lifetime of different approaches.

We use several key system parameters to model the long-term energy efficiency of a sensor system with network programming. The CPU utilization is denoted as f_{cpu}. The radio duty cycle is denoted as f_{radio}. The average working period between two reprogramming actions is denoted as t. The sum of reprogramming energy consumption and the execution energy consumption during t is

$$E_t = v(f_{radio}I_{radio} + f_{cpu}I_{cpu})t + E_{reprog} \tag{8}$$

where v is the voltage, I_{radio} is the current draw when the radio is active, I_{cpu} is the current draw when the CPU is active, E_{reprog} is the energy consumption during reprogramming.

Batteries exhibit self-discharge phenomenon. The energy consumption due to self-discharge can be calculated as follows:

$$B'(L) = rLB \tag{9}$$

where r is the self-discharge rate (% per day), L is the time in terms of days, and B = 2200 mA h is the battery capacity. Considering standard NiMH batteries lose half their charge after one year, we assume $r = \frac{0.5}{365}$.

We can get the following equation:

$$B = E_t \cdot \frac{L}{t + c_0 \cdot S(m)} + B'(L) \tag{10}$$

The lifetime of the network can thus be calculated as follows.

$$L(t) = \frac{B}{\frac{E_t}{t+c_0 \cdot S(m)} + rB} \tag{11}$$

Note that, we assume time for loading the code is sufficiently small compared to t, hence E_t roughly represents the energy consumption during t plus the time for dissemination, $c_0 \cdot S(m)$.

Table 6
Platform-related parameters.

Parameter	Value
v	3 V
I_{radio}	20 mA [23]
I_{cpu}	0.5 mA [23]
c_0	200 s/KB
e_s	0.00104 mJ/byte
e_l	0.0293 mJ/rela
e_f	0.000587 mJ/byte

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

We measure all the platform-related parameters on the TelosB nodes with MSP430 processors. The results are summarized in Table 6.

As sensor networks typically work under low radio cycles, the radio duty cycle here is set to 0.02. It means that the radio is active for 2% of the time while is in sleep mode for 98% of the time. We compare three reprogramming approaches (all under SenSpire OS) with varied reprogramming intervals (i.e., the time between two reprogramming actions).

- Full image replacement approach. In this approach, applications are statically linked with the kernel to form a single image. This approach requires a relative large transferred file size. We set the size as 20 KB. The CPU utilization is set to 0.05.
- Modular approach (e.g., SenSpire OS + SELF [19]). This approach reduces the transferred file size because only individual modules need to be transferred. We set the size as 2 KB. Compared to the full image replacement approach, this approach adds indirection cost, so the CPU utilization will slightly increase. We observed that for wide range of applications, the CPU slow down compared to the full image approach is very small. For example, for the 11 benchmarks, we observed that the CPU slow down falls with 0.005. We further look into the program code and find that inter-module communications happens infrequently in a period, e.g., limited to 10 system calls in these benchmarks. We measured that a typical task in these benchmarks executes nearly 300 μs, and one indirection causes approximately 2 μs extra overhead. Hence, the increase in CPU utilization is small. We conservatively consider $f_{cpu} = 0.05 + \frac{2\times10}{300} \leqslant 0.057$ for the modular approach.
- Virtual machines (e.g., Maté [7]). This approach has the smallest transferred file size because VM instructions can express higher level functionalities. We set the size as 200 bytes. However, each instruction are interpreted at run-time and cause an extra overhead. We conservatively consider the VM approach causes a three times slow down: $f_{cpu} = 0.15$.

Table 7
Application-related parameters for three reprogramming approaches.

	Full image	Module	VM
$S(m)$	20,000	2000	200
$S(flash)$	20,000	1500	200
N_{rela}	0	50	0
f_{radio}	0.01	0.01	0.01
f_{cpu}	0.05	0.057	0.15

Table 7 summarizes the application-related parameters for the three approaches.

Fig. 16 shows how the network lifetime relates to reprogramming intervals for the three reprogramming approaches. We can see that the VM approach is only energy efficient with short reprogramming intervals, e.g., one or two days. When the program is expected to run for a long time, the VM approach is not energy efficient because the virtual instructions are inefficient to execute. The full image replacement approach consumes a large fraction of energy during code dissemination, hence is less energy efficient than both approaches when reprogramming interval is short (e.g., $\leqslant$17). With the reprogramming interval increases, the network lifetime for the full image replacement approach and the modular approach would be similar. Overall, the modular approach is the most energy efficient within a reasonable reprogramming interval (e.g., 5–30 days). In real-world deployments, reprogramming is usually infrequent, e.g., approximately once every month in GreenOrbs [24]. In this case, the modular

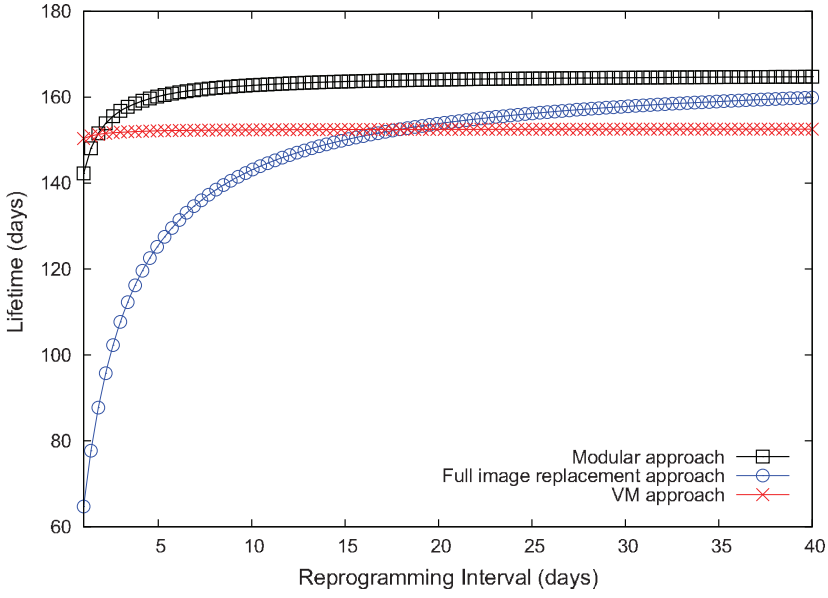


Fig. 16. Comparison of three reprogramming approaches in terms of network lifetime.

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

approach is the most energy efficient as can be seen from Fig. 16.

To summarize, we have the following observations: (1) When the reprogramming interval increases, the energy cost during reprogramming becomes less important while the energy cost during execution becomes more important. (2) As expected, the modular approach and the native approach will have a longer lifetime with a long reprogramming interval since the native code is more efficient to execute compared with VM. (3) The VM approach will be less attractive with a long reprogramming interval, especially for CPU-intensive applications.

4. Related work

Modular design [15,14,19] and network reprogramming [4,6,25] are two closely related topics for networked embedded systems such as sensor networks [1,2].

The modular design approach can effectively reduce the transferred code size, hence improves energy efficiency of network reprogramming.

ELF is one of the most common object code format for dynamic linking. It is a standard format for object files and executables that is used for most modern Unix-like systems. An ELF object file include both program code and data and additional information such as a symbol table, the names of all external unresolved symbols, and relocation tables. The relocation tables are used to locate the program code and data at other places in memory than for which the object code originally was assembled. One problem with the ELF format is the overhead in terms of bytes to be transmitted across the network, compared to pre-linked modules. There are a number of reasons for the extra overhead.

Therefore, we design SELF to have a good tradeoff between the module size and the module's functionalities (e.g. for enabling flexible programming, fine grained configuration). At the high level, we maintain ELF's basic structure, e.g. the data, bss, text, sym, reloc sections while reducing unnecessary overheads for resource-constrained sensor nodes. The drawback of the SELF format is that it requires a special compressor utility is for creating the SELF files. It is also worth mentioning that the tailored file format (SELF) is only used for energy-efficient dissemination. Developers can still use the compiled ELF format as inputs to inspect the detailed information. In addition, we have implemented tools for conversion from SELF to ELF.

In the following, we review some notable modular designs in sensor OSes.

Modular design for TinyOS. FlexCup [26] supports update of binary components in TinyOS. Compared to our approach, FlexCup's loading overhead is large because it makes extensive use of flash to perform linking, and requires a hardware reboot to execute a new application. As illustrated in [27], hardware reboot causes lost of application states, which can be costly to recover again. Recently, TinyLD supports dynamic loading for applications based on TOSThreads [28]. TinyLD is still in the process of development, hence it does not address module dependencies and module communications in a systematical manner.

Modular design for Contiki OS. Contiki OS [8,14] supports both ELF and CELF. The CELF only redefines long data types to short ones, hence CELF is not yet optimized. In particular, the Contiki kernel symbols occupy a large fraction of module size. FiGaRo [29] handles module dependencies based on Contiki OS. It focuses on the programming aspect, and does not address other issues like module format optimization, inter-module communications, and analysis of energy consumption. Minilink [17] is a recent work that enables stateful mobile modules for sensor network. Minilink incorporates several techniques to reduce the module file size, including eliminating common prefix in the symbol table, placing the relocation entries in the code section, etc. Compared to SELF, Minilink enables stateful migration of code. This is achieved at additional cost of serialization of relevant data. On the other hand, SELF has included some other reduction techniques, including the chained reference technique, the use of system call gate. In addition, SELF enables fine-grained module configuration. We have also proposed a network lifetime model for better understanding of various reprogramming approaches.

Modular design for SOS. SOS [15] natively supports dynamically-loadable modules (MELF). MELF adopts many techniques to reduce the module size. Programming based MELF is more demanding in comparison to SELF. First, MELF needs a special SOS module header to be defined by programmers. Second, MELF requires global variables to be declared in a global data structure. Third, direct communications between MELF modules can only be via function IDs, instead of function names. In addition, MELF does not handle module dependencies.

Modular design for LiteOS. LiteOS [21] also supports dynamic loadable modules. It utilizes a differential patching idea with a mathematical model. However, it requires a training process at the PC side to ensure the correctness of the model. However, it remains challenging how many training data should be sufficient to ensure system reliability.

In fact, modular design has other benefits (other than reduce data size for efficient reprogramming). Future network embedded systems will probably be developed by different teams. Modular design will make things easier for code reuse and cooperation among multiple teams.

Yi et al. [5] describe other updating strategies, e.g., incremental update and function update. In incremental update [9,11], the delta between two successive versions of codes is generated and disseminated. In function update, the modified functions and the metadata for re-linking need to be disseminated. In general, incremental update and function update (if applied to modules) will further reduce the dissemination cost during reprogramming if small modifications are made. There are circumstances, however, module-level update is more beneficial, e.g. when many functions in a module are modified [5]. This is because incremental update and function update need more metadata. Another important issue is that both incremental update and function update need additional steps to reconstruct the code for execution. For example, in incremental update, the new code needs to be reconstructed using the old code and the received delta. In function update, relocation needs to be performed to ensure the correctness of function calls.

Compared to existing modular designs in sensor OSes. Our approach is more sophisticated in reducing the

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007

module size, handling module dependencies and inter-module communications.

There is a tradeoff between the transferred file size and the loading efficiency. For example, data encoding and decoding [30] can be used to further reduce the module size, but it incurs a large decoding overhead during the loading process. To investigate whether the energy efficiency of different techniques, we have further developed a long-term energy efficiency model. This model considers different aspects of reprogramming and also battery discharge characteristics. With this model, we can get a deep insight into how different techniques impact the overall energy efficiency and network lifetime.

5. Conclusions

In this paper, we present a systematic modular design approach for networked embedded systems. We effectively reduce the module file size to enable efficient network reprogramming, while at the same time retain necessary information to maintain module flexibility. We further handle module dependencies in a fine-grained manner, which improves system reliability while keeping the system configuration to its minimum requirement.

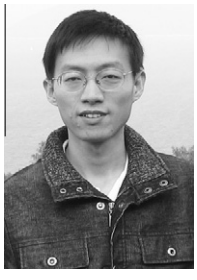
We have implemented the modular approach based on a micro embedded OS, SenSpire OS, for AVR and MSP430 platforms. The evaluation results show that the proposed SELF module file format is 4.6–7.6 times smaller than the standard ELF format, and is 1.6–2.4 times smaller than the CELF format. SELF retains necessary information to enable flexible modular programming and inter-module communications. We have further developed a long-term energy efficiency model. This model considers different aspects of reprogramming and also battery discharge characteristics. With this model, we can predict the lifetime of sensor nodes using different techniques.

Acknowledgement

This work is supported by the Fundamental Research Funds for the Central Universities (2012QNA5007), the National Science Foundation of China (Grant No. 61070155 and Grant No. 61202402), and the Program for New Century Excellent Talents in University (NCET-09-0685).

References

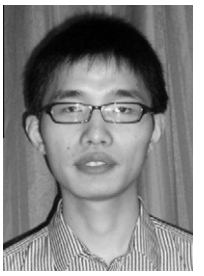
[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, Wireless sensor networks: a survey, Computer Networks 38 (2002) 393–422.
[2] L. Mo, Y. He, Y. Liu, J. Zhao, S. Tang, X.-Y. Li, G. Dai, Canopy closure estimates with greenorbs: sustainable sensing in the forest, in: Proceedings of ACM SenSys, 2009.
[3] A. Bonivento, C. Fischione, L. Necchi, F. Pianegiani, Alberto Sangiovanni-Vincentelli, System level design for clustered wireless sensor networks, IEEE Transactions on Industrial Informatics 3 (3) (2007) 202–214.
[4] Q. Wang, Y. Zhu, L. Cheng, Reprogramming wireless sensor networks: challenges and approaches, IEEE Network Magazine 20 (3) (2006) 48–55.
[5] S. Yi, H. Min, Y. Cho, J. Hong, Adaptive multilevel code update protocol for real-time sensor operating systems, IEEE Transactions on Industrial Informatics 4 (4) (2008) 250–260.
[6] J. W. Hui, D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale, in: Proceedings of ACM SenSys, 2004.
[7] P. Levis, D. Culler, Maté: a tiny virtual machine for sensor networks, in: Proceedings of ASPLOS, 2002.
[8] A. Dunkels, N. Finne, J. Eriksson, T. Voigt, Run-time dynamic linking for reprogramming wireless sensor networks, in: Proceedings of ACM SenSys, 2006.
[9] J. Jeong, D. Culler, Incremental network programming for wireless sensors, in: Proceedings of IEEE SECON, 2004.
[10] P. von Richenbash, R. Wattenhofer, Decoding code on a sensor node, in: Proceedings of IEEE DCOSS, 2008.
[11] R.K. Panta, S. Bagchi, Hermes: fast and energy efficient incremental code updates for wireless sensor networks, in: Proceedings of IEEE INFOCOM, 2009.
[12] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, M. Welsh, Fidelity and yield in a volcano monitoring sensor network, in: Proceedings of USENIX OSDI, 2006.
[13] K. Langendoen, A. Baggio, O. Visser, Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture, in: Proceedings of WPDRTS, 2006.
[14] A. Dunkels, B. Grönvall, T. Voigt, Contikia lightweight and flexible operating system for tiny networked sensors, in: Proceedings of EmNets, 2004.
[15] C.-C. Han, R. Kumar, R. Shea, E. Kohler, M. Srivastava, A dynamic operating system for sensor nodes, in: Proceedings of ACM MobiSys, 2005.
[16] W. Dong, C. Chen, X. Liu, Y. Liu, J. Bu, K. Zheng, SenSpire OS: a predictable, flexible, and efficient operating system for wireless sensor networks, IEEE Transactions on Computers 60 (12) (2001) 1788–1801.
[17] R.K. Moritz Strübe, K. Stengel, M. Daum, F. Dressler, Stateful mobile modules for sensor networks, in: Proceedings of IEEE DCOSS, 2010.
[18] J.R. Levine, Linkers and Loaders, Morgan Kaufmann, 2000.
[19] W. Dong, C. Chen, X. Liu, J. Bu, Y. Liu, Dynamic linking and loading in networked embedded systems, in: Proceedings of IEEE MASS, 2009.
[20] A. Griffith, GCC: The Complete Reference, McGraw-Hill, 2004.
[21] Q. Cao, T. Abdelzaher, J. Stankovic, T. He, The LiteOS operating system: towards unix-like abstractions for wireless sensor networks, in: Proceedings of ACM/IEEE IPSN, 2008.
[22] W. Dong, C. Chen, X. Liu, J. Bu, Y. Liu, Performance of bulk data dissemination in wireless sensor networks, in: Proceedings of IEEE DCOSS, 2009.
[23] R. Fonseca, P. Dutta, P. Levis, I. Stoica, Quanto: tracking energy in networked embedded systems, in: Proceedings of USENIX OSDI, 2008.
[24] Y. Liu, Y. He, M. Li, J. Wang, K. Liu, L. Mo, W. Dong, Z. Yang, M. Xi, J. Zhao, X.-Y. Li, Does wireless sensor network scale? A measurement study on GreenOrbs, in: Proceedings of IEEE INFOCOM, 2011.
[25] R.K. Panta, I. Khalil, S. Bagchi, Stream: low overhead wireless reprogramming for sensor networks, in: Proceedings of IEEE INFOCOM, 2007.
[26] P.J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, K. Rothermel, FlexCup: a flexible and efficient code update mechanism for sensor networks, in: Proceedings of EWSN, 2006.
[27] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, J. Regehr, Surviving sensor network software faults, in: Proceedings of ACM SOSP, 2009.
[28] K. Klues, C.-J.M. Liang, J. yeup Paek, R. Musaloiu-E., P. Levis, A. Terzis, R. Govindan, TOSThreads: safe and non-invasive preemption in TinyOS, in: Proceedings of ACM SenSys, 2009.
[29] L. Mottola, G.P. Picco, A.A. Sheikh, FiGaRo: fine-grained software reconfiguration for wireless sensor networks, in: Proceedings of EWSN, 2008.
[30] N. Tsiftes, A. Dunkels, T. Voigt, Efficient sensor network reprogramming through compression of executable modules, in: Proceedings of IEEE SECON, 2008.



Wei Dong received the BS and Ph.D. degrees in Computer Science from Zhejiang University in 2005 and 2010, respectively. He was a Postdoc Fellow at the Department of Computer Science and Engineering in Hong Kong University of Science and Technology from Dec. 2010 to Dec. 2011. He is currently an assistant professor at the College of Computer Science in Zhejiang University. His research interests include networked embedded systemsand wireless sensor networks.

Chun Chen received his Bachelor of Mathematics degree from Xiamen University, China, in 1981, and his Masters and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1984 and 1990 respectively. He is a professor in College of Computer Science, the Dean of College of Software, and the Director of Institute of Computer Software at Zhejiang University. His research activity is in image processing, computer vision, CAD/CAM, CSCW, and embedded system.

Jiajun Bu received the BS and Ph.D. degrees in Computer Science from Zhejiang University, China, in 1995 and 2000, respectively. He is a professor in College of Computer Science and the deputy dean of the Department of Digital Media and Network Technology at Zhejiang University. His research interests include embedded system, mobile multimedia, and data mining. He is a member of the IEEE and the ACM.



Chao Huang received his BS degree from Dalian University of Technology. He is currently a M.phil. student in the College of Computer Science of Zhejiang University. His research interests include networked embedded systems and wireless sensor networks.

Please cite this article in press as: W. Dong et al., Enabling efficient reprogramming through reduction of executable modules in networked embedded systems, Ad Hoc Netw. (2012), http://dx.doi.org/10.1016/j.adhoc.2012.07.007