Elon: Enabling Efficient and Long-Term Reprogramming for Wireless Sensor Networks

Wei Dong[†‡], Yunhao Liu[‡⋆], Xiaofan Wu[†], Lin Gu[‡], and Chun Chen[†]
[†]Zhejiang Key Laboratory of Service Robot, College of Computer Science, Zhejiang University
[‡]Hong Kong University of Science and Technology
[⋆]Tsinghua National Laboratory for Information Science and Technology
{dongw, wuxf, chenc}@zju.edu.cn, {liu, lingu}@cse.ust.hk

Abstract

We present a new mechanism called Elon for enabling efficient and long-term reprogramming in wireless sensor networks. Elon reduces the transferred code size significantly by introducing the concept of *replaceable* component. It avoids the cost of hardware reboot with a novel software reboot mechanism. Moreover, it significantly prolongs the reprogramming lifetime by avoiding flash writes for TelosB nodes. Experimental results show that Elon transfers up to 120–389 times less information than Deluge, and 18–42 times less information than Stream. The software reboot mechanism that Elon applies reduces the rebooting cost by 50.4%–53.87% in terms of beacon packets, and 56.83% in terms of unsynchronized nodes. In addition, Elon prolongs the reprogramming lifetime by a factor of 2.3.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design—*Distributed networks*; D.4.7 [**Operating Systems**]: Organization and Design

General Terms

Design, Experimentation, Performance

Keywords

Wireless Sensor Network, Reprogramming, Component, Reboot

1 Introduction

Wireless sensor networks (WSNs) have been studied for a wide range of applications, such as ecological surveillance, habitat monitoring, infrastructure protection, etc. WSN applications often need to be changed after deployment for a variety of reasons—upgrading node software, correcting software bugs, and patching security holes. Many large-scale WSNs, however, are deployed in environments where physically collecting previously deployed nodes is either very difficult or infeasible. Enabling sensor nodes to be reprogrammable over the air is a crucial technique to address such challenges [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMETRICS'10, June 14–18, 2010, New York, New York, USA.
Copyright 2010 ACM 978-1-4503-0038-4/10/06 ...$5.00

There are several key factors that affect the reprogramming efficiency and reprogramming lifetime of a WSN, including the transferred code size, the loading cost, and the reprogramming voltage requirement.

First, the transferred code size greatly impacts the transmission overhead. To make a WSN reprogrammable, Deluge [2] disseminates the entire program image, including the application code, the TinyOS kernel [3], and the reprogramming protocol. Hence, it must handle a large code size even for a simple application. For example, in the TinyOS 2.1 distribution [3], the `Blink` application with Deluge support consumes more than 35 KB code size. Such a code size results in a long transmission time and much energy consumption during dissemination. Stream [4] pre-installs the reprogramming protocol on the external flash. Consequently, for simple applications, it reduces the code size significantly. Specifically, for the `Blink` application in TinyOS 2.1, Stream needs approximately 9 KB transferred code size. Unfortunately, Stream is still far from sufficient for real-world and complex applications. For example, for the `TestNetwork` application in TinyOS 2.1, Stream would need more than 35 KB code size for reprogramming. This is because the `TestNetwork` application contains many TinyOS kernel services like CTP [5], Drip [6].

Second, the loading cost for executing the new code needs to be minimized. In particular, we should try to avoid hardware reboot as it wastes energy as well as dropping data. A system normally stores many state data such as routing tables [5], synchronization tables [7], dissemination keys [6], etc. After rebooting, some nodes may take hours to fully come back online. For example, in a recent deployment on Reventador Volcano, reboots from a software error led to 3-day network outage, reducing mean node uptime from >90% to 69% [8, 9].

Third, it is beneficial to avoid flash writes during reprogramming as writing to flash requires a much higher voltage than the minimum operational voltage for the commonly used TelosB nodes. The MSP430F1611 microcontroller for the TelosB nodes can operate down to 1.8 V [10], while the minimum required voltage during a flash write or erase is 2.7 V [11]. If the voltage falls below 2.7 V during a write or erase, the result of the write or erase will be unpredictable [11]. As a result, based on study on the battery discharge curve [12], network reprogramming has to be performed in the starting 23% duration of the entire network lifetime, significantly impairing the reprogrammability of existing reprogramming protocols that rely heavily on flash writes [2, 4, 13, 14].

To address the above three key issues, we present Elon, a holistic mechanism based on the TinyOS operating system [3]. Elon reduces the transmission overhead of both the TinyOS kernel services and the reprogramming protocol by defining replaceable component that needs to be reprogrammed. Elon extends the nesC compiler to provide the boundary between replaceable component and

the TinyOS kernel. The replaceable component is defined by replaceable code and replaceable data which are allowed to be declared by applications. The replaceable component can be replaced or updated in the future. Applications should also wire TinyOS kernel components (e.g., CTP [5], Drip [6], FTSP [7]) that would probably be used in the future. The intuition behind Elon's design is that for real-world complex applications, the TinyOS kernel components that will be used could be identified in the base version. The application logic is mainly controlled by the application component. Therefore, by defining part of the application component as "replaceable", we can reprogram the replaceable component in the future without the need of disseminating the TinyOS kernel which is typically the largest and most complex part of a program image.

Moreover, Elon allows applications to define "system" interfaces (via the "@system()" annotation) for accessing TinyOS kernel services (similar to syscalls in Unix-like systems). These interfaces define the boundary between the replaceable component and the TinyOS kernel. For downcalls (i.e., TinyOS commands) defined in the system interfaces, we relocate the references (i.e., command calls in the replaceable component) in the *updated* version to the corresponding addresses (i.e., command implementations in kernel components) in the *base* version. For upcalls (i.e., TinyOS events) defined in the system interfaces, we keep a system jump table to redirect the references (i.e., event calls/signals in kernel components) in the *base* version to the corresponding addresses (i.e., event handler implementations in the replaceable component) in the *updated* version.

By isolating TinyOS kernel components from the replaceable component that needs to be reprogrammed, Elon is able to reduce the transferred code size significantly. Isolation also allows Elon to reboot a node in a fine-grained software manner: Elon only reboots the replaceable component without losing TinyOS kernel data. Moreover, by placing the replaceable component on RAM (for Von-Neumann architectures), we avoid flash writes, extending the reprogramming lifetime significantly for the commonly used TelosB nodes.

We implement Elon on TinyOS 2.1 for TelosB nodes, and conduct comprehensive experiments to examine its performance. Evaluation results show that (i) Elon effectively reduces the transferred code size: it transfers up to 120–389 times less information than Deluge, and 18–42 less information than Stream. (ii) Elon significantly reduces the rebooting cost for core TinyOS components: for CTP [5] and Drip [6], it reduces the cost by 53.87% and 50.4%, respectively, in terms of beacon packets; for FTSP [7], it reduces the cost by 56.83% in terms of unsynchronized nodes. (iii) Elon extends the reprogramming lifetime by a factor of 2.3 for TelosB nodes compared to existing reprogramming approaches. (iv) the overhead of Elon is acceptably small for real-world complex applications in terms of programmer cost, memory overhead, and execution overhead.

The rest of this paper is structured as follows. Section 2 provides the necessary background. Section 3 presents the design details. Section 4 presents the evaluation results. Section 5 describes related work most pertinent to our work. Section 6 discusses limitations and our future work. Finally, Section 7 concludes this paper.

2 Background

Elon builds on top of the TinyOS operating system [3]. This section provides the necessary background on this system (Section 2.1, 2.2), core sensor network services (Section 2.3, 2.4), and a real-world and complex application (Section 2.5) that motivates our design.

2.1 TinyOS

TinyOS [3] is the standard operating system for the sensor network platforms. To enable a flexible architecture and a low resource consumption, TinyOS programming is based on components which are wired together at design-time to create a single TinyOS-App image. Like objects, components couple code and data. Unlike objects, however, they can only be instantiated at compile time [8]. TinyOS components have interfaces which define commands (downcalls) and events (upcalls). Components interact in two directions, i.e., one component can *use* commands provided by another component; also, one component can *signal* events to another component [15].

The TinyOS core has a highly restricted, purely event-driven execution model which consists of interrupts and tasks. Interrupts execute at a higher priority and can preempt the execution of tasks. Tasks execute at a lower priority and are scheduled in a FIFO manner. TinyOS code marked *async* could be possibly executed in the interrupt context while the default *sync* code can only be executed in the task context. TinyOS tasks are written in a run-to-completion manner. They cannot be preempted by one another, or self-suspended. For this reason, I/O operations are performed in a split-phase manner, i.e., an I/O request call initiates the operation and returns with status information before the corresponding completion callback is invoked when the I/O operation is indeed finished. TinyOS tasks and interrupts do not retain stack frames while inactive, they share a common system stack.

2.2 nesC/GCC Toolchain

TinyOS components are written in a dialect of C called nesC [16]. nesC exposes a programming model that incorporates event-driven execution, a flexible concurrency model, and component-oriented application design [16]. Restrictions on the programming model allow the nesC compiler to perform whole-program analysis, including dead code elimination (which optimizes the code size), aggressive function inlining (which optimizes the execution efficiency), etc. These optimizations, however, complicate Elon's implementation. This issue will be further elaborated in Section 3.2.

The nesC compiler compiles TinyOS/nesC code into C code which is further processed by the GCC toolchain. First, the C compiler (cc) compiles C files into object files (.o). The object file not only contains the code binary but also contains metadata, such as symbol tables, relocation tables, string tables, etc. The existence of the metadata allows the references in code and data to be fixed in later stages (in the linking stage or the loading stage) because the actual addresses for these references are not necessarily known at compile time. Second, the linker (ld) combines multiple object files into an executable file. In TinyOS, the loading addresses must be determined prior to the linking stage. The generated executable file cannot be loaded to different addresses or linked with other files. This is because the linker performs relocation and linking at the linking stage, after which the relocation tables are usually removed away. Both the object file and the executable file are in ELF format [17], which is a standard object code format for most modern Unix-like systems. Finally, the executable file is further transformed into a target format, e.g., Intel hex (.ihex) or Motolora S Record (.srec). The hardware in-system-programmer loads the file (ihex or srec) onto the program flash in the sensor node for execution.

2.3 Deluge, Stream

Deluge [2] is the standard reprogramming protocol distributed with TinyOS. Deluge enables complete system reprogramming, i.e., the whole TinyOS-App image can be replaced by a new TinyOS-App image. In order to reprogram a network of sensor nodes for multiple times, the new TinyOS-App image must include

the reprogramming protocol. Deluge stores the new code on the external flash during code dissemination. It uses a small bootloader, i.e., TOSBoot, for loading the new code from the external flash onto the program flash. Finally, it forces a hardware reboot to execute the new code.

Stream [4] reduces the transferred code size by pre-installing the reprogramming protocol on the external flash as another code image (i.e., the reprogramming image). The application only needs to include a lightweight reprogramming support component which is responsible for rebooting to the reprogramming image when reprogramming is needed. After receiving the new application code, the reprogramming image reboots again to the new application code (the application image). Although Stream reduces the transferred code size significantly for simple applications, it is still insufficient for real-world and complex applications because the kernel components (such as CTP [5], Drip [6], FTSP [7]) still need to be disseminated. Moreover, image switching using hardware reboot degrades the reprogramming reliability because, according to our own experience, keeping a network in a consistent state (i.e., all nodes are in the reprogramming state or application state) is very difficult due to unreliable wireless links.

2.4 CTP, Drip, FTSP

Most existing reprogramming mechanisms force a hardware reboot for executing the new code [2, 4, 13, 14]. However, reboot is not free. Computing systems maintain state to provide useful services. Rebooting a node clears this state and recovering this state can be costly for many important TinyOS kernel components [8].

CTP [5] is a collection tree protocol that collects data to a sink. A CTP node maintains two tables. The link estimation table stores the ETX estimate of each link. The routing table stores routing candidates and their corresponding path-ETX estimates to the sink. Rebooting a node causes a CTP node to lose information in these tables. Recovering this information is costly, either taking time to collect (for the link estimation table), or incurring a significant communication overhead for sending routing beacons (for the routing table) due to the use of Trickle timers [18].

Drip [6] is a dissemination protocol for small data items. Drip stores the values of all data items in a RAM cache. Rebooting a Drip node causes a node to lose all values of the data items, which need to be collected again from other nodes. The use of Trickle timers [18] in Drip causes a significant increase of beacons after the reboot.

FTSP [7] is a time synchronization protocol that establishes a global time over the network. An FTSP node stores a table of sync beacons from other synchronized nodes to infer the relationship between its local time and the global time by linear regression. Rebooting a node clears this table, causing the rebooting node temporally unsynchronized.

2.5 GreenOrbs

GreenOrbs [19, 20] is a recently deployed wireless sensor network that aims at achieving long-term kilo-scale surveillance in the vast forest. The GreenOrbs uses TelosB nodes, and builds on top of the TinyOS operating system. The GreenOrbs program includes the CTP component [5] for collecting multiple types of sensor data, e.g., light, temperature, humidity, to a sink. To increase the flexibility, GreenOrbs includes the Drip component [6] for disseminating key system parameters, e.g., the duty cycle and transmission power settings. To achieve energy efficiency, GreenOrbs also includes the FTSP component [7] for enabling synchronous low duty cycling. In the current implementation, each node works three minutes every hour (i.e., 5% duty cycle). The development of the GreenOrbs system motivates our work. First, deploying a large number of sensor nodes into the forest is very costly. Therefore, network reprogramming is an essential requirement for such a system. Second, existing mechanisms on TinyOS [2, 4] incur a large transferred code size for the GreenOrbs application, which is time-consuming and energy-inefficient to disseminate. Third, rebooting a node is costly, because after rebooting, a node loses all system states and needs to be awake until it is synchronized and fully functional again. Finally, the maximum voltage of the 2200mAh NiMH rechargeable battery that GreenOrbs currently uses is 2.8 V. Because the energy consumption during the initial deployment can be quite large (all nodes must stay awake for synchronization before entering the working mode), the voltage of some nodes quickly falls below 2.7 V, which makes most existing reprogramming mechanisms [2, 4, 13, 14, 21] useless during the majority of the node lifetime.

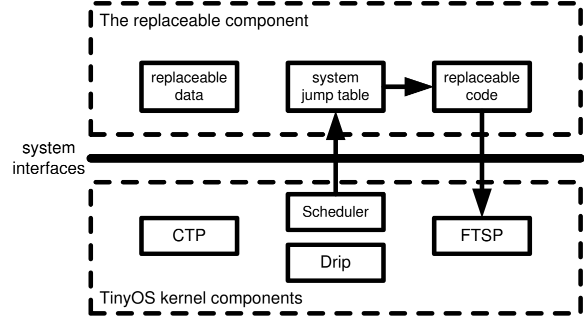

Figure 1: Elon's extensions to TinyOS. An application can specify replaceable component and TinyOS kernel components. System interfaces are boundaries between the replaceable component and TinyOS kernel components. The replaceable component calls system commands by address relocation. TinyOS kernel components signal system events by indirections via a system jump table.

3 Design

This section describes Elon's design to address three reprogramming issues outlined in Section 1. Figure 1 illustrates Elon's extensions to TinyOS. TinyOS kernel components (such as CTP [5], Drip [6], FTSP [7]) should be included in the program of the base version for future use. For the base version, both kernel components and the replaceable component are programmed onto the program flash by the hardware in-system-programmer. A node works correctly after a hardware reboot. For updated versions, only the replaceable component is generated and disseminated to each sensor node. When a node receives the new code, it uses a software reboot mechanism to execute the new code.

Figure 2 shows Elon's extensions to the nesC/GCC toolchain for TinyOS applications. The application and kernel components are in the nesC language. Applications can specify *replaceable* and *system* annotations. The modified nesC compiler processes these annotations and generates the C code (.c) along with a system jump table in assembly (.s). These source files (.c and .s) are further processed by the compiler (cc) and assembler (as) in the GCC toolchain. The outputs is comprised of a set of object files (.o). For the base version, we use a two-phase linking process to generate the executable ELF file. For updated versions, we have developed a tool (`relocate.exe`) to generate the relocated ELF file (for network reprogramming).

The next three subsections detail Elon's designs. Section 3.1 describes how Elon identifies and places the replaceable component. Section 3.2 describes how Elon identifies and implements system

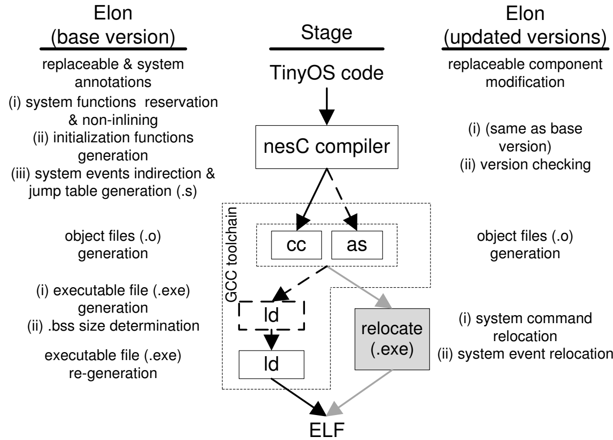

Figure 2: Elon's extensions to the nesC/GCC toolchain. The process for generating the program of the base version is shown on the left, and the process for generating programs of the updated versions is shown on the right. Dashed boxes and arrows indicate additional steps for generating the base version. Gray boxes and arrows indicate additional steps for generating the updated versions.

interfaces. Section 3.3 describes how Elon reboots a node to execute the code.

3.1 Replaceable Component

Applications need to define the replaceable component that needs to be reprogrammed in the future. The replaceable component consists of the replaceable code, the replaceable data, and a system jump table. The replaceable code and replaceable data are declared by the application via the "@replaceable()" annotation. The system jump table is generated automatically by the modified nesC/GCC toolchain. Applications can specify the replaceable functions (code) and replaceable variables (data) separately. Applications can also specify an entire TinyOS component as replaceable. In this case, all variables and functions defined in this component default to replaceable unless they are explicitly specified as non-replaceable. The code below shows the use of the annotation in details:

```
module BlinkToCntLedsC {
  // ...
}
implementation {
  uint16_t counter @replaceable();
  void Boot.booted() @replaceable();
  void Timer.fired() @replaceable();
  // ...
}
```

In the above example, the variable `counter` and the functions `Boot.booted()`, `Timer.fired()` are labelled as replaceable. Replaceable data and replaceable code are placed in separate sections, i.e., .vdata for initialized replaceable data, .vbss for unintialized replaceable data, and .vtext for replaceable code. The existence of .vbss section helps reduce the transferred code size further because the initial values for the .vbss section do not need to be transferred (the values are known to be zero). The automatically generated system jump table is also placed in a separate section—the .jmptab section. We must provide the base addresses for these sections to the linker. Before we elaborate on the details of determining the base addresses, we first give an overview of the memory layout on TelosB nodes (shown in Table 1).

Table 1: Memory organization of TelosB nodes [22]

Memory		
Memory Main: interrupt vector Main: code memory	Size Flash Flash	48KB 0FFFFh – 0FFE0h 0FFFFh – 04000h
RAM (Total)	Size	10KB 038FFh – 01100h
Extended	Size	8KB 038FFh – 01900h
Mirrored	Size	2KB 018FFh – 01100h
Information memory	Size Flash	256 Byte 010FFh – 01000h
Boot memory	Size ROM	1KB 0FFFh – 0C00h
RAM (mirrored at 018FFh - 01100h)	Size	2KB 09FFh – 0200h
Peripherals	16-bit 8-bit 8-bit SFR	01FFh – 0100h 0FFh – 010h 0Fh – 00h

3.1.1 Memory Layout

The TelosB node has 10 KB data RAM (from address 0x1100 to address 0x38FF) and 48 KB internal program flash (from address 0x4000 to address 0xFFFF). The highest 32 bytes of code space (from address 0xFFE0 to address 0xFFFF) is reserved for storing the interrupt vectors.

The nesC compiler compiles the TinyOS nesC code to C code, which is further compiled and linked to an executable ELF by the GCC toolchain. The ELF file is used for linking and loading in traditional Unix-like systems. However, the standard ELF loader is usually missing for current sensor nodes. Therefore, the ELF file is further transformed to a simplified format, e.g., Intel hex (.ihex) or Motolora S Record (.srec). In the in-system-programming process, based on the the ihex file or srec file, the hardware in-system-programmer writes necessary sections to the program flash, including the .text section (for the program code), the .data section (for initializing the .data section in RAM), and .vectors section (for interrupt vectors). When the code starts execution, it first executes an initializer (usually starts at address 0x4000) which is responsible for initializing the .data section and the .bss section in RAM. It then jumps to the `main()` function for further initialization, e.g., initializing the system stack, the registers, TinyOS components, etc. Figure 3(a) illustrates the final memory layout after this process.

3.1.2 Address Determination

Determining the base addresses of the .vdata, .vbss, .vtext, and .jmptab sections should satisfy two requirements. First, it should not conflict with other sections. Second, it should make efficient use of memory.

We place these sections as follows. First, the .vdata section and the .vbss section are placed in RAM. The .vdata section is directly after (in higher-address region than) the .bss section, and the .vbss section is directly after the .vdata section. Second, the .vtext section is placed in the program flash for the base version (serves as a golden image), and is placed in RAM for updated versions (in order to avoid flash writes during network reprogramming). For the base version, the .vtext section is directly after the .data section in the program flash. For updated versions, the .vtext section is directly after the .vbss section. Third, the .jmptab section is *always* placed in the start of RAM (i.e., 0x1100 for TelosB nodes) because

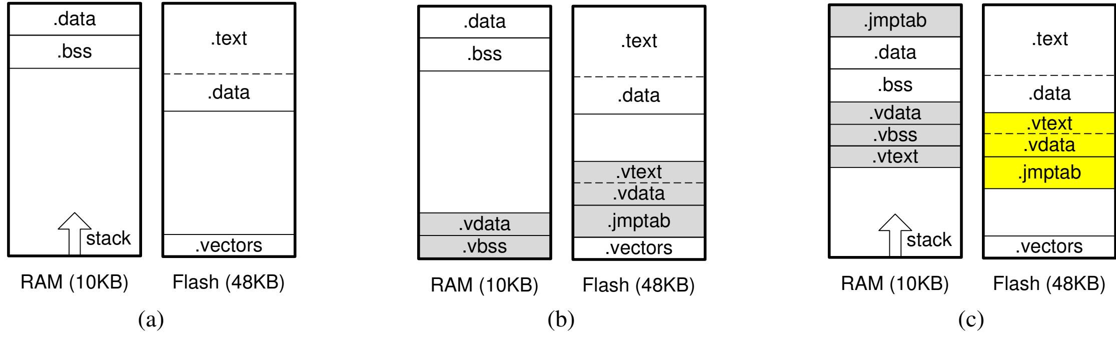

Figure 3: Memory layout in RAM and the program flash. (a) memory layout for TinyOS. (b) memory layout for Elon after the first phase of linking. (c) memory layout for Elon after the two-phase linking process (the yellow boxes indicate the golden image).

the system jump table needs to be located at a fixed address, and it needs to be modifiable.

The base addresses of these sections should be specified to the linker prior to the linking stage. This means that in order to place the current section, we must obtain the sizes of all preceding sections before the linking stage (or after the compilation stage). However, the section sizes in the final executable file cannot be determined until the linking process is done. The reason is that the linker has to combine multiple object files to generate the final executable file [23]. For example, the section sizes of .text, .data, and .bss cannot be determined after compilation if the application links additional external libraries. To address this issue, we adopt a two-phase linking process. We first specify *non-conflict* base addresses to the linker as follows:

$$\begin{aligned} \text{START}_{.\text{vbss}} &= \text{END}_{\text{RAM}} - \text{SIZE}_{.\text{vbss}} \\ \text{START}_{.\text{vdata}} &= \text{START}_{.\text{vbss}} - \text{SIZE}_{.\text{vdata}} \\ \text{START}_{.\text{jmptab}} &= \text{END}_{\text{flash}} - \text{SIZE}_{.\text{vectors}} - \text{SIZE}_{.\text{jmptab}} \\ \text{START}_{.\text{vtext}} &= \text{START}_{.\text{jmptab}} - \text{SIZE}_{.\text{vtext+.vdata}} \end{aligned} \tag{1}$$

where $\text{SIZE}_{.\text{vectors}}$ is a constant for a microcontroller (e.g., $\text{SIZE}_{.\text{vectors}} = 32$ for MSP430F1611 used by TelosB nodes), and $\text{SIZE}_{.\text{vbss}}$, $\text{SIZE}_{.\text{vdata}}$, $\text{SIZE}_{.\text{jmptab}}$, $\text{SIZE}_{.\text{vtext+.vdata}}$ are all known after compilation because the corresponding sections are defined in a single compilation unit by prohibiting external libraries from being replaceable (note that the nesC compiler compiles all .nc files into a single C source file, app.c). Figure 3(b) illustrates the memory layout after this linking stage. After the first phase of linking, we obtain all the section sizes. Then we use it to determine the *final* base addresses:

$$\begin{aligned} \text{START}_{.\text{jmptab}} &= \text{START}_{\text{RAM}} \\ \text{START}_{.\text{data}} &= \text{START}_{.\text{jmptab}} + \text{SIZE}_{.\text{jmptab}} \\ \text{START}_{.\text{bss}} &= \text{START}_{.\text{data}} + \text{SIZE}_{.\text{data}} \\ \text{START}_{.\text{vdata}} &= \text{START}_{.\text{bss}} + \text{SIZE}_{.\text{bss}} \\ \text{START}_{.\text{vbss}} &= \text{START}_{.\text{vdata}} + \text{SIZE}_{.\text{vdata}} \\ \text{START}^{\text{updated}}_{.\text{vtext}} &= \text{START}_{.\text{vbss}} + \text{SIZE}_{.\text{vbss}} \\ \text{START}^{\text{golden}}_{.\text{vtext}} &= \text{START}_{\text{Flash}} + \text{SIZE}_{.\text{text+.data}} \\ \text{START}^{\text{golden}}_{.\text{jmptab}} &= \text{START}^{\text{golden}}_{.\text{vtext}} + \text{SIZE}_{.\text{vtext+.vdata}} \end{aligned} \tag{2}$$

Figure 3(c) illustrates the final memory layout after this two linking process. Note that, we reserve a golden image at the program flash for error recovery in the face of failure.

3.2 System Interface

System interfaces are the boundaries between the replaceable component and TinyOS kernel components. Applications need to customize the TinyOS kernel by wiring kernel components that will be used in the future. The TinyOS kernel components will be installed onto the nodes for the first time and will not incur additional dissemination overhead during reprogramming. Applications need to identify system interfaces, declared via the "@system()" annotation, for accessing kernel services. The code below shows the use of the annotation in details:

```
module BlinkToCntLedsC {
  uses interface Boot @system();
  uses interface Timer<TMilli> @system();
  // ...
}
```

In the above example, both the `Boot` interface and the `Timer<TMilli>` interface are declared as system interfaces.

TinyOS interfaces contain commands (downcalls) and events (upcalls). For system interfaces, the commands are provided by the kernel components and represent the non-replaceable part that does not need reprogramming; on the other hand, the events should be implemented by the application component and usually represent the replaceable part that needs reprogramming.

3.2.1 Modification to the nesC Compiler

We need to address two issues in the nesC compilation process. First, the system commands and events should not be inlined because the caller part and callee part reside in different sections (i.e., the .text section and the .vtext section respectively). However, the current nesC compiler performs aggressive inlining for optimizing the execution efficiency. To address this issue, we modify the nesC compiler to make system commands and system events as "noninline". Second, system commands and events should not be removed even if they are not used in the base version because they will probably be used in the future. However, the current nesC compiler does dead code elimination for optimizing the code size. To address this issue, we modify the nesC compiler to make these functions as "extern" in order to reserve them in the base version.

3.2.2 Relocation and Indirection

For the program of the base version, the GCC toolchain relocates references to variables and functions correctly at link time. However, for any updated versions that will be reprogrammed onto the nodes through the network, changes in the source code make the address relocation incorrect. For system commands, the GCC toolchain will relocate their addresses within the updated version.

This is incorrect as we only reprogram the replaceable component of the updated version onto the nodes, references to system calls in the replaceable component should be relocated to corresponding addresses in the kernel components of the base version (that exist on the nodes). For example, it is possible that a system command `Timer.startPeriodic()` is located at 0x5000 in the base version while located at 0x5010 in the updated version. When generating an updated version, relocating references to 0x5010 by the GCC toolchain will be incorrect because 0x5000 is the actual address of the system call that is already programmed onto the nodes during the initial in-system-programming process. Also, for system events, it is often the case that their addresses in the updated version are different than the ones in the base version. Therefore, references to system events in the kernel component should point to corresponding addresses in the updated version. To address this issue, we adopt two approaches—relocation and indirection, for system commands and system events, respectively.

First, for system commands (also for the public kernel data), when generating the program of the updated version (i.e., the replaceable component), we relocate all references to system commands to the corresponding addresses in the base version. We develop a tool, `relocate.exe`, to do this kind of relocation. The command below shows the use of the tool in details:

```
relocate.exe -d start_vdata -b start_vbss \
             -t start_vtext -r main.exe main.o
```

where the base addresses for the .vdata, .vbss, and .vtext sections (i.e., `start_vdata`, `start_vbss`, `start_vtext`) are determined by Eq. (2), `main.exe` is the executable file of the base version, and `main.o` is the object file of the updated version. The `relocate.exe` program parses the relocation entries in `main.o` and relocates the references in `main.o` to system commands implemented in `main.exe`. `main.o` is also the output file.

Second, for system events, we maintain a system jump table to redirect references in the base version to corresponding addresses in the updated version. The relocation technique for system commands described above does not apply here because we cannot modify the program code of the base version. For each system event that is replaceable, we allocate one entry in the system jump table. The code below shows the system jump table for the `Boot` system interface and the `Timer<TMilli>` system interface:

```
br #BlinkToCntLedsC$Boot$booted
br #BlinkToCntLedsC$Timer$fired
```

Invocation to these system events must be via the system jump table entries which redirect them to the actual implementations. When the addresses of system events change in the updated version, references in the base version do not change because the system jump table is placed at a fixed location in RAM (i.e., the starting address of RAM, described in Section 3.1). Instead, we modify the system jump table entries in RAM to reflect this change. The command below shows how to generate a correct system jump table for the updated version:

```
relocate.exe -t 1100 -r main.o jmptab.o
```

We use `relocate.exe` again for relocating system jump table entries (`jmptab.o`) to the corresponding addresses in the program of the updated version (`main.o`). In the above command, 1100 (with hexadecimal base) denotes the starting address in RAM on TelosB nodes. In this way, we retain access to system events we are interested in in updated versions.

3.3 Reboot

There are two methods for programming a node, i.e., in-system-programming, and network reprogramming. In the final stage of in-system-programming, the system executes a hardware reboot procedure. This hardware reboot procedure is also executed when the reboot button is pressed or the power is on. In the final stage of network reprogramming, we use a software reboot procedure in order to reduce the rebooting cost.

3.3.1 Hardware Reboot

In the final stage of in-system-programming, the system executes a hardware reboot procedure which is responsible for initialization for the data, system stack, the registers, TinyOS components, etc. Because Elon adds specific sections to the generated ELF file, we need to modify the default initializer generated by the GCC toolchain. The modified initialization procedure is as follows:

1. Copy the default system jump table from the program flash to the start of RAM.
2. Initialize the .data section on RAM (i.e., copy the .data section from the program flash to RAM).
3. Initialize the .bss section on RAM to zero.
4. Initialize the .vdata section on RAM (i.e., copy the .vdata section from the program flash to RAM).
5. Initialize the .vbss section on RAM to zero.
6. Jump to the `main()` function which includes initialization of the system stack, the registers, and TinyOS components, etc.

Figure 3(c) shows the final memory layout after this procedure. There are two things that worth mentioning here. First, we program the replaceable component of the base version onto the program flash. This provides a golden image for error recovery in the face of failure. Second, we reduce the cost of copying the .vtext section to RAM by generating a default system jump table with entries pointing to corresponding locations in the program flash.

3.3.2 Software Reboot

In the final stage of network reprogramming, we use a software reboot procedure in order to reduce the rebooting cost.

To avoid the use of external flash and make efficient use of RAM, we load the replaceable data and replaceable code onto RAM during code dissemination. A question in our design is whether the RAM can accommodate the replaceable code. Our experience in developing the GreenOrbs application demonstrate that it suffices for a large number of software changing scenarios [19]. We will discuss this issue further in Section 6. Loading the new code onto RAM will incur two severe safety-related problems. First, the current program will be corrupted. For example, the current program may handle a system `Timer.fired()` event periodically. During reprogramming, however, the implementation of `Timer.fired()` function could be partly re-written. When this event is signaled, the system crashes. To address this issue, we re-write all valid system jump table entries to point to a dummy event handler before reprogramming. Second, it may fail to receive a complete new program. To address this issue, we provide a rollback mechanism. We rollback to the golden image.

When we complete receiving and loading the replaceable data and replaceable code, we start a software reboot procedure to restart the node. To summarize, Elon goes through the following steps for reprogramming a node:

1. Write the system jump table entries to point to a default dummy event hander.
2. Receive data, and write replaceable data and replaceable code to RAM.

3. If the entire replaceable component is received correctly, go to step 5, else go to step 4.
4. Write the "golden" replaceable data to RAM. (the replaceable code will be located on the program flash for execution in the face of failure.)
5. Initialize the .vbss section on RAM to zero.
6. Write the system jump table: if the replaceable component is received correctly, write the newly received system jump table, otherwise write the "golden" default system jump table on the program flash.
7. Jump to the `main()` function which includes initialization of the system stack, the registers, TinyOS components, etc.

Compared to hardware reboot, the software reboot procedure does not include initialization for the kernel data. This is important for keeping system states persistent across reboots.

4 Evaluation

This section evaluates how Elon improves sensor network reprogramming efficiency (in terms of transferred code size and rebooting cost) and reprogramming lifetime. We also evaluate the overhead Elon introduces. Section 4.1 describes the evaluation methodology. Section 4.2 shows evaluation results in terms of transferred code size, and its impact on dissemination time and transmission overhead. Section 4.3 shows evaluation results in terms of rebooting cost. Section 4.4 shows evaluation results in terms of reprogramming lifetime. Finally, Section 4.5 shows Elon's overhead in terms of programmer cost, memory overhead, and execution overhead, respectively.

4.1 Methodology

All experiments use TelosB nodes. To evaluate the performance of Elon, we consider the following software change scenarios for TinyOS applications.

1. BlinkToCntLeds: we change the `Blink` application to the `CntToLeds` application. `Blink` is an application that toggles one LED periodically, and `CntToLeds` is an application that displays the lowest three bits of the counting sequence on the LEDs.
2. BlinkToCntRfm: we change the `Blink` application to the `CntToLedsAndRfm` application. `CntToLedsAndRfm` is an application that transmits the counting sequence over the radio and displays the lowest three bits of the counting sequence on the LEDs.
3. GreenOrbs: we consider a real-world software change scenario in the development of GreenOrbs [19]. In the program of the base version, each node periodically reports sensor data to a collection sink by CTP; we want to reprogram an updated version in which each node additionally reports diagnostic packets to give more visibility into the system.

For network experiments, we use a testbed of 10 TelosB nodes in a linear structure. We set the transmission power level to 3 to emulate multihop transmissions. To get the statistics, we wrote two components, i.e., `RamLogC` and `ExtLogC`, to log aggregated data in RAM, and to log system events in the external flash. In order to get the timing information, there is a sync node that periodically broadcasts sync beacons to synchronize all other nodes at the maximum transmission power. For synchronization, we use the packet-level time synchronization interface provided by TinyOS, which can achieve synchronization accuracy in less than 1 ms [24]. After the experiments, we gather the data through one-hop wireless links.

Table 2: Comparison of transferred code size for Deluge, Stream, and Elon (in bytes).

	Deluge	Stream (lower bound)	Elon
BlinkToCntLeds	32008	2674	82
BlinkToCntRfm	32182	11608	266
GreenOrbs	>48 KB	47640	2440

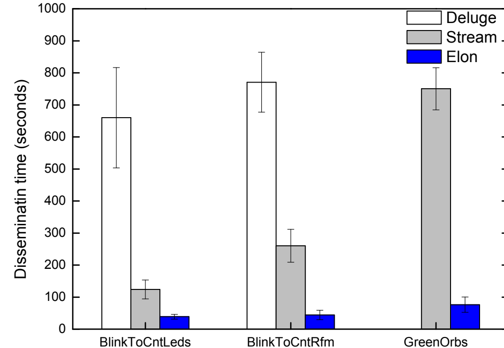

Figure 4: Comparison of dissemination time (in seconds) for Deluge, Stream, and Elon.

4.2 Transferred Code Size

Table 2 shows the transferred code sizes for three reprogramming approaches, i.e., Deluge [2], Stream [4], and Elon, for the software change cases mentioned above. As the Stream code is not available for TinyOS 2.1, we use a lower bound for comparison (i.e., applications with Stream support do not wire the Stream support component). We can see that the code sizes for Deluge are very large because applications with Deluge support wire the DelugeC component which consumes a large fraction of code size. For the GreenOrbs application, Deluge cannot be used simply because the program size with Deluge support exceeds the maximum program size for TelosB nodes. For reprogramming simple applications, e.g., BlinkToCntLeds and BlinkToCntRfm, Stream reduces the code size significantly. This is because Stream does not include the reprogramming protocol. Instead, it pre-installs it on the external flash as another code image. However, for reprogramming complex applications, e.g., GreenOrbs, the transferred code size for Stream is still very large because a large number of kernel components (such as CTP, Drip, FTSP) are wired in the application and need to be transferred during reprogramming. Elon reduces the overhead of kernel components by identifying replaceable component. From the figures shown in Table 2, we can see that Elon reduces the transferred code size significantly compared to Deluge and Stream: it transfers 120–389 times less information than Deluge, and 18–42 less information than Stream.

A small transferred code size translates to smaller dissemination time and smaller transmission overhead, which is important for energy-constrained sensor networks. To see the impact of the transferred code size on the dissemination time and the transmission overhead, we disseminate these applications through the Deluge protocol. As mentioned above, we obtain the timing information by packet-level synchronization provided in TinyOS, and we get the number of packet transmissions by use of the `RamLogC` component. Figure 4 compares the dissemination times of Deluge, Stream, and Elon. The result of GreenOrbs with Deluge support is not available because the program size exceeds the maximum program size for TelosB nodes, hence cannot be compiled correctly.

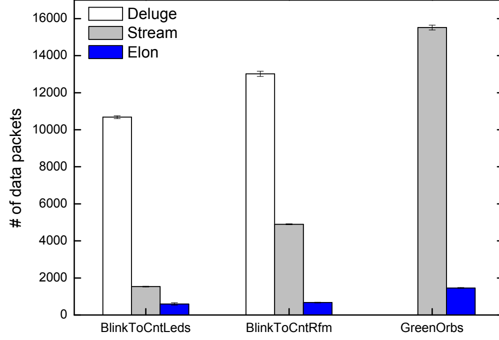

Figure 5: Comparison of total number of packet transmissions for Deluge, Stream, and Elon.

Table 3: Hardware rebooting time and software rebooting time for the updated applications (ms).

	TinyOS	Elon
BlinkToCntLeds	20.5	16.93
BlinkToCntRfm	23.3	17.25
GreenOrbs	87.6	83.2

As expected, Elon outperforms Deluge and Stream significantly, especially for complex applications with a large number of kernel components. For the two relatively simple applications, i.e., BlinkToCntLeds and BlinkToCntRfm, Elon is 16.05–16.39 times faster than Deluge, and 2.2–4.87 times faster than Stream. For the complex GreenOrbs application, Elon is 8.82 times faster than Stream. This illustrates that replaceable component identification and isolation are very important for reprogramming real-world and complex applications. Figure 5 compares the total number of packets transmitted by all nodes in the network for Deluge, Stream, and Elon. Like dissemination time, Elon reduces the transmission overhead significantly compared to other approaches. As indicated by the GreenOrbs application, avoiding transmissions of kernel components in Elon results in a very large savings in the transmission overhead.

4.3 Rebooting Cost

The rebooting cost of Elon is low compared to existing reprogramming approaches. First, we eliminate the cost of flash I/O operations which incur extra energy consumptions on current sensor nodes. For example, a flash I/O operation on TelosB nodes consume 5–12 mA, which is much larger than 0.5 mA energy consumption when the CPU is active. In Deluge, after a hardware reboot, the TOSBoot copies the program code from the external flash to the the program flash. Stream incurs even more flash I/O operations due to image switching. Elon does not have this overhead. Second, software reboot is faster than hardware reboot because re-initialization of kernel data is not needed. Table 3 compares the hardware reboot time and the software reboot time. We can see that Elon's software reboot mechanism is 5%–35% faster than the hardware approach. Finally and most importantly, Elon does not lose kernel data caused by a hardware reboot. To see the impact of losing kernel data, we conduct experiments running typical TinyOS services, e.g., CTP [5], Drip [6], and FTSP [7].

For CTP [5], the cost of hardware reboots is the increase of routing beacons. Figure 6 shows the total accumulated number of beacons for the CTP protocol during a total of 2000 seconds. We compare the results when there are no-reboots, hardware reboots,

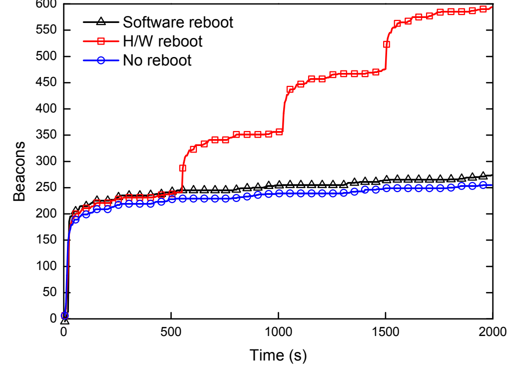

Figure 6: Comparison of the total number of beacons in CTP for hardware reboots, software reboots, and no-reboots.

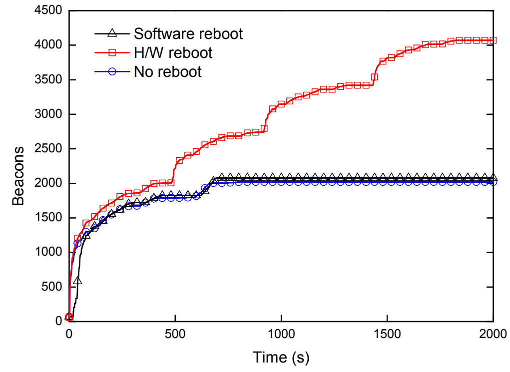

Figure 7: Comparison of the total number of beacons in Drip for hardware reboots, software reboots, and no-reboots.

and software reboots. We use the `RamLogC` component for logging the accumulated number of beacons at each individual node. This information is transmitted to the sink by CTP. For no-reboots, the total number of beacons is 255; for hardware reboots, the total number of beacons increases to 594; for software reboots, the number of beacons is 274. The cost of hardware reboots is high because CTP uses a Trickle timer [18] to control the frequency of beaconing rate. When a node reboots, it will have a high frequency of beaconing rate in order to find new routes. Elon's software reboot mechanism keeps system states persistent across reboots. Compared to hardware reboots, Elon reduces the rebooting overhead by 53.87% in terms of beacon packets.

For Drip [6], the cost of hardware reboots is also the increase of beacon packets. Figure 7 shows the total accumulated number of beacons for the Drip protocol disseminating 32 keys during a total of 2000 seconds. We compare the results when there are no-reboots, hardware reboots, and software reboots. Similarly, we use the `RamLogC` component for logging purpose. After the experiment, we collect the data individually for each node. As Drip also uses the Trickle timer [18] to control the beaconing rate, similar results can be obtained. For no-reboots, the total number of beacons is 2020; for hardware reboots, the total number of beacons increases to 4191; for software reboots, the number of beacons is 2079. Compared to hardware reboots, Elon reduces the rebooting overhead by 50.4% in terms of beacon packets.

For FTSP [7], the cost of hardware reboots is the decrease in the number of synchronized nodes. Figure 8 shows the CDF of the

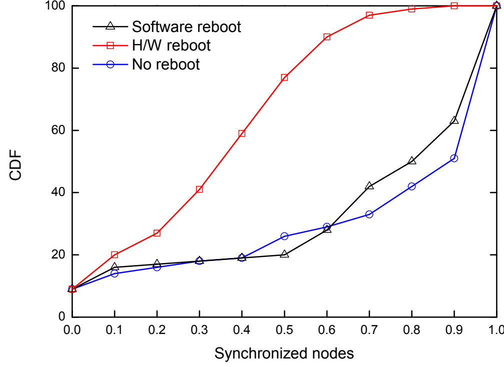

Figure 8: CDF of synchronized nodes in FTSP when there are hardware reboots, software reboots, and no-reboots.

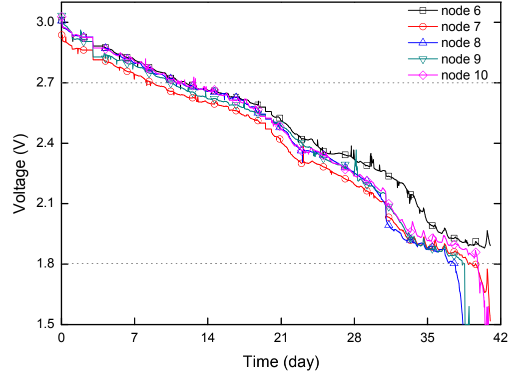

Figure 9: Voltage readings for five typical nodes running the CTP protocol with default TinyOS LPL MAC.

number of synchronized nodes by querying the system periodically for a total of 600 times, when there are no-reboots, hardware reboots, and software reboots. When there are reboots, we reboot the nodes every two minutes with a random startup time. Packet losses and other real-world variations cause non-zero amount of unsynchronized nodes even in the no-reboot case. Without reboots, there are approximately 74.52% synchronized nodes on average. With hardware reboots, there are approximately 35.05% synchronized node on average due to relatively high rebooting rate. With software reboot, there are approximately 71.96% synchronized nodes on average. Compared to hardware reboot, Elon's software reboot mechanism reduces the rebooting overhead by 56.83% in terms of unsynchronized nodes.

4.4 Reprogramming Lifetime

To investigate the reprogramming lifetimes of different approaches, we deployed an indoor prototype system that runs the basic functionality of the GreenOrbs application. This version of program uses CTP to collect the voltage and other sensor data from multiple nodes. In this version, we do not include the FTSP component for fast implementation. To enable energy efficiency, we use the default low power listening (LPL) MAC in TinyOS. We use a 20 seconds data reporting rate and a 250 ms sleep interval. A 250 ms sleep interval translates to 4.4% duty cycle when there is no traffic, with 11 ms check interval in the implementation of TinyOS. The actual duty cycle in our case should be much higher than 4.4% due to high data reporting rate (the LPL MAC waits for an additional duration of 100 ms when a packet is received). We successfully run the application for approximately 40 days.

Figure 9 shows the voltage readings for five typical nodes. We can see that the voltage decreases from 3 V to 1.8 V, during which the system operates correctly. This is because TelosB nodes have a minimum voltage of 1.8 V for program execution [10]. However, flash writes require a much higher voltage to ensure reliability. Specifically, flash writes on TelosB requires at least 2.7 V as documented in the MSP430 users' guide [11]. If the voltage falls below 2.7 V during a write or erase, the result of the write or erase will be unpredictable [11]. This means that, in this application scenario, the reprogramming lifetime is only 11 days for existing approaches that require flash writes (including Deluge [2], Stream [4], FlexCup [13], etc). Elon extends the reprogramming lifetime to 37 days (or more), by a factor of 2.3.

4.5 Overhead

This section evaluates Elon's overhead in term of programmer cost, memory overhead, and execution overhead, respectively.

Table 4: Number of TinyOS kernel components in the base version.

	Deluge	Elon	increase
BlinkToCntLeds	4	4	0%
BlinkToCntRfm	4	7	75%
GreenOrbs	25	25	0%

4.5.1 Programmer Cost

First, Elon requires programmers to identify TinyOS kernel components in the base version for future use. Table 4 compares the number of TinyOS kernel components for Deluge and Elon. We do not show the results for Stream as the results are similar to Deluge. For the software change scenario of BlinkToCntLeds, there is no increase. For the software change scenario of BlinkToCntRfm, the increase is 75%. This is because TinyOS kernel components (such as radio) are not used in the base version, but they will be used in the updated version. Therefore, they should be additionally wired in the application. For the software change scenario of GreenOrbs, however, no additional kernel components should be wired because the program of the base version already uses a large number of kernel components. This results indicate that for complex and well designed applications, the number of additional kernel components should be small because a lot of kernel components should be already used in the base version.

Second, Elon requires programmers to specify replaceable and system annotations. Depending on reprogramming needs, the number of annotations can be different. However, each annotation requires a simple @replaceable() and @system() code addition. Therefore, the cost of annotations is small.

4.5.2 Memory Overhead

We examine the Elon's memory overhead in terms of RAM consumption and program flash consumption, respectively. There are several factors that impact the Elon's memory overhead. First, we place replaceable code on RAM, which increases RAM consumption and decreases program flash consumption. Second, we need to include the reprogramming support component, which increases program flash consumption. Third, we specify some TinyOS interfaces as "system", which may increase program flash consumption (system commands and events are included in even if they are not referenced in the base version) or decrease program flash consumption (system functions are non-inlined).

Table 5 shows the RAM consumption for the programs of the base version and Table 6 shows the RAM consumption for the pro-

Table 5: RAM consumption in bytes for the programs of the base version.

	TinyOS	Elon	increase
BlinkToCntLeds	35	310	785.7%
BlinkToCntRfm	35	554	1482%
GreenOrbs	6326	7820	23.6%

Table 6: RAM consumption in bytes for the programs of the updated version.

	TinyOS	Elon	increase
BlinkToCntLeds	37	384	937.8%
BlinkToCntRfm	320	796	148.7%
GreenOrbs	6426	8146	26.8%

Table 7: Program flash consumption in bytes for the programs of the base version.

	TinyOS	Elon	increase
BlinkToCntLeds	2522	6836	171%
BlinkToCntRfm	2522	15820	527.2%
GreenOrbs	47414	47494	0.17%

Table 8: Program flash consumption in bytes for the programs of the updated version.

	TinyOS	Elon	increase
BlinkToCntLeds	2642	6836	158.7%
BlinkToCntRfm	11576	15820	36.6%
GreenOrbs	47640	47494	−0.3%

grams of the updated version. For the two simple cases, i.e., BlinkToCntLeds and BlinkToCntRfm, the relative increases in RAM consumption are quite large. However, the absolute values are sufficiently small compared to a total of 10 KB RAM on TelosB nodes. For the complex GreenOrbs case, the relative increase in RAM consumption drops down to 23.6%–26.8%.

Table 7 shows the program flash consumption for the programs of the base version and Table 8 shows the program flash consumption for the programs of the updated version. We can see that, for the two simple software change scenarios, i.e., BlinkToCntLeds and BlinkToCntRfm, the increase is large because the reprogramming support component consumes a fixed code size of approximately 4 KB. However, for the complex case, i.e., GreenOrbs, the increase is very small for the base version (0.17%), and is even negative for the updated version (−0.3%). The reason for the negative increase is because the updated replaceable component is placed in RAM, instead of in the program flash.

4.5.3 Execution Overhead

Elon slightly increases the execution overhead because of two reasons. First, the system annotation forces all system commands to be non-inlined, hence there is an additional cost compared to the case when the functions can be inlined. Second, the system annotation forces all system events to be non-inlined and redirected via a system jump table. This is an additional cost.

We use a Tektronix TDS3034C oscilloscope to measure the execution time of a TinyOS task. We instrument the `TinySchedulerC` component to provide such information. Table 9 shows the task execution times for the programs of the base version, and Table 10 shows the task execution times for the programs of the updated version. We did not show the results for the GreenOrbs application because it involves a large number of TinyOS tasks which complicates the measurement. From the figures in Tables 9–10, we can see that the increase is small, especially for relative complex tasks.

Table 9: Task execution times for the programs of the base version (μs).

	Task	TinyOS	Elon	increase
BlinkToCntLeds	Timer.fired	273	296	8.4%
BlinkToCntRfm	Timer.fired	273	306	12.0%

Table 10: Task execution times for the programs of the updated version (μs).

	Task	TinyOS	Elon	increase
BlinkToCntLeds	Timer.fired	308	323	4.8%
BlinkToCntRfm	Timer.fired	1640	1653	0.79%
	sendDone	98.4	105	6.7%
	receive	260	262	0.77%

5 Related Work

Reprogramming wireless sensor networks has been an active research area in recent years [1, 2, 25, 4, 26, 27, 28, 14]. Several systems, such as Mate [29] and VM* [30], provide virtual machines that run on resource-constrained sensor nodes. They allow for dissemination of a small code size because the virtual machine code is much more compact than the native code. However, the execution of virtual machine code is much less efficient than native code, which severely degrades the energy efficiency of the deployed sensor nodes when the program is expected to be executed for a long time. Also, the virtual machine code is less expressive than native code.

TinyOS [3] is the standard operating system for sensor networks. TinyOS does not support loadable modules. An application is compiled with TinyOS to form a single TinyOS-App image. Deluge [2] provides reprogramming support for TinyOS applications. As sensor nodes need to be reprogrammed for multiple times, Deluge transfers the single TinyOS-App image along with the reprogramming protocol. After a node receives a new TinyOS-App image, the TinyOS bootloader (i.e., TOSBoot) copies the code from the external flash to the program flash, and it forces a hardware reboot to execute the new code. Stream [4] reduces the transferred code size by pre-installing the reprogramming protocol on the external flash as another application image (i.e., the reprogramming image). During the reprogramming process, each node first reboots to the reprogramming image for retrieving the new application code. When the new application code is received, each node reboots again to the new application. Stream reduces the code size significantly for simple applications. However, it is still insufficient for real-world and complex applications which usually include a large number of kernel components. Moreover, Stream uses hardware reboots to switch between the reprogramming image and the application image. Therefore, it incurs the cost of hardware reboot and more importantly, according to our own experience, keeping a network in a consistent state (i.e., all nodes are in the reprogramming state or the application state) is very difficult because of unreliable wireless links.

The SOS [31], Contiki [32, 21], and the FlexCup extension [13] to TinyOS support loadable modules. In these systems, individual modules can be loaded dynamically on the nodes. Specific challenges exist in these systems. First, they require disseminating symbol tables and relocation tables for linking and relocating.

These may be quite large, typically 45% to 55% of the object file [33, 14]. Second, they make extensive use of flash. Therefore, they cannot reprogram a sensor network when the voltage of certain nodes falls below 2.7 V (but above 1.8 V) for the commonly used TelosB nodes [11], which limits the reprogramming lifetime.

Incremental reprogramming [34, 33, 35, 14, 36] is a technique to reduce the transferred code size by disseminating a program difference. Such research is complementary to our work. We can generate the incremental changes for the replaceable component to further reduce the code size. The work of compressing the program code [37] and disseminating with network coding [38, 39, 28] is also complementary to our work. We can further use these techniques to reduce the transferred code size and the dissemination cost.

Reconfigurability can also be achieved by disseminating parameters understood by the program of the base version through Drip [6], Dip [40], or DHV [41]. Although this approach is lightweight (e.g., the transferred code size is small, and it does not require a minimum voltage of 2.7 V), it requires all parameters to be well defined in the base version, which is hard to design in the early stage of software developments. With this technique, the application logic cannot be modified when there are no corresponding parameters pre-defined in the base version. Elon is much more flexible: both the replaceable data and the replaceable code can be modified once they have been identified.

6 Discussion

A basic observation to reduce the transferred code size is to disseminate what is needed, in other words, the application. Similar to the design concept of loadable modules, our approach identifies the replaceable component that needs to be reprogrammed. With this technique, common kernel components (that do not need to be reprogrammed) do not need to be disseminated. The traditional dynamic linking and loading technique [23, 21] for loadable modules is not appropriate for our scenario because of two reasons. First, it requires sophisticated OS support (e.g., the ELF loader) which is currently unavailable in TinyOS. Second, it incurs additional overhead, e.g., symbol tables and relocation tables. With a simplified design, we are able to provide a mechanism with minimal OS support and smaller transferred code size. We trade off less flexibility in the program layouts that can be slightly different on different sensor nodes. However, different program layouts can be easily avoided by programming the same binary code in the in-system-programming process.

For the Mica series node, our approach cannot prolong the reprogramming lifetime because of two reasons. First, the Mica series node uses the Atmega128L microcontroller which is based on the Harvard architecture. Therefore, the program code cannot be placed in data RAM for execution. Second, the minimum operational voltage is 2.7 V (instead of 1.8 V), i.e., nodes die once the voltage falls below 2.7 V. Another limitation of the Mica series node is that the RAM size is only 4 KB. Our approach, however, is still useful by placing the replaceable code on the program flash. It is still effective in reducing the transferred code size, and, at the same time, avoiding the cost of hardware reboot.

The more recent Telos node uses a low-power MSP430F1611 microcontroller which is based on the Von-Neumann architecture. The MSP430F1611 microcontroller can operate down to 1.8 V. However, the minimum voltage during a flash write or erase operation is still 2.7 V. If the voltage falls below 2.7 V during a write or erase, the result of the write or erase will be unpredictable [11]. Therefore, it is beneficial to avoid flash writes during reprogramming so as to prolong the reprogramming lifetime. Our approach avoids flash writes by placing replaceable component on RAM.

Today, even with many new offerings in microcontrollers, the MSP430F1611 remains a competitive choice for sensor platform design [42]. Our design illustrates that avoiding flash writes is beneficial for low-power microcontrollers (that can operate down to 1.8 V), because most low-power flash requires at least 2.5 V–2.7 V for writing or erasing, e.g., the program flash of MSP430F161x, ST M25P80 (external flash for the TelosB node) [43], AT45DB041B (external flash for the MicaZ node) [44], etc.

A question in our design is whether we can accommodate the replaceable code in RAM which is a precious resource for sensor nodes. Our experience in developing the GreenOrbs application using TelosB nodes demonstrates that it suffices for many software change scenarios because of two reasons. First, the data RAM consumption is usually small for TinyOS applications ($\leq$ 4 KB) partly because TinyOS was originally designed for Mica series nodes with 4 KB RAM. However, TelosB nodes have 10 KB RAM. Second, the total size of the replaceable components for well-designed applications is usually small ($\approx$ 2 KB for the GreenOrbs application). If the size of the replaceable code does exceed the available RAM size, we can simply place it on the program flash. In this case, we require a higher voltage that ensures reliable flash writes.

TinyOS's event-driven model has simplified the design of Elon's software reboot mechanism. First, TinyOS kernel does not keep application data. Second, TinyOS does not have blocking I/O operations. Therefore, rebooting a node does not require hacking into the TinyOS kernel for preserving application data, cancelling blocking I/O operations, or halting threads. We do need to protect the system jump table to avoid jumping to corrupted memory locations during the reprogramming process.

Compared to complete system reprogramming [2, 4], Elon limits the reprogrammability to the replaceable component. It suffices for most cases because the application behavior is mainly controlled by the application logic, which can be specified as replaceable. It should also be noted that the "@replaceable" annotation is at the granularity of individual variables and individual functions, instead of a "component". Therefore, Elon also allows update to key parameters and key functions in TinyOS kernel components once they have been identified as replaceable. On the other hand, compared to system parameters reconfiguration, we improve the reprogrammability because we allow updates to the native code.

Ensuring a safe update to the program of the base version is an important issue that we have not covered thus far. In Elon's current implementation, we have developed a simple version checking mechanism: it is not allowed to change the set of kernel components and system interfaces. In the future, we would like to develop a declarative language [45] for updating programs, which can avoid modifying the source code directly and can protect against occasional errors introduced by programmers.

7 Conclusion

This paper presents a new mechanism called Elon for enabling efficient and long-term reprogramming in wireless sensor networks. Elon reduces the transferred code size significantly by introducing the concept of *replaceable* component. It avoids the cost of hardware reboot with a novel software reboot mechanism. Moreover, it significantly prolongs the reprogramming lifetime by avoiding flash writes for TelosB nodes. Experimental results show that Elon transfers up to 120–389 times less information than Deluge, and 18–42 times less information than Stream. The software reboot mechanism that Elon applies reduces the rebooting cost by 50.4%–53.87% in terms of beacon packets, and 56.83% in terms of unsynchronized nodes. In addition, Elon prolongs the reprogramming lifetime by a factor of 2.3. The overhead of Elon is acceptably small for real-world applications in terms of programmer cost, memory overhead, and execution overhead.

Acknowledgments

The authors would like to thank Gong Chen, Yi Gao for their help in Elon's implementation and evaluation. We thank all members in the GreenOrbs project (http://www.greenorbs.org) for their contributions to this work. This work is supported by the National Basic Research Program of China (973 Program) under grant No. 2006CB303000, and NSFC/RGC Joint Research Scheme N_HKUST602/08.

8 References

[1] Q. Wang, Y. Zhu, and L. Cheng, "Reprogramming Wireless Sensor Networks: Challenges and Approaches," *IEEE Network Magazine*, vol. 20(3), pp. 48–55, 2006.

[2] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of ACM SenSys*, 2004.

[3] TinyOS. [Online]. Available: http://www.tinyos.net

[4] R. K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," in *Proceedings of IEEE INFOCOM*, 2007.

[5] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," in *Proceedings of ACM SenSys*, 2009.

[6] G. Tolle and D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," in *Proceedings of EWSN*, 2005.

[7] M. Maróti, B. Kusy, G. Simon, and Á. Lédeczi, "The Flooding Time Synchronization Protocol," in *Proceedings of ACM SenSys*, 2004.

[8] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, "Surviving Sensor Network Software Faults," in *Proceedings of ACM SOSP*, 2009.

[9] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and Yield in a Volcano Monitoring Sensor Networks," in *Proceedings of USENIX OSDI*, 2006.

[10] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research," in *Proceedings of ACM/IEEE IPSN*, 2005.

[11] MSP430x1xx Family User's Guide (Rev. F). [Online]. Available: http://focus.ti.com/lit/ug/slau049f/slau049f.pdf

[12] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel, "Meeting Lifetime Goals with Energy Levels," in *Proceedings of ACM SenSys*, 2007.

[13] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks," in *Proceedings of EWSN*, 2006.

[14] R. K. Panta and S. Bagchi, "Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks," in *Proceedings of IEEE INFOCOM*, 2009.

[15] W. Dong, C. Chen, X. Liu, and J. Bu, "Providing OS Support for Wireless Sensor Networks: Challenges and Approaches," *IEEE Communications Surveys and Tutorials*, vol. 12(4), 2010, to appear.

[16] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: A holistic approach to networked embedded systems," in *Proceedings of ACM PLDI*, 2003.

[17] TIS Committee, *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2*, 1995. [Online]. Available: http://refspecs.freestandards.org/elf/elf.pdf

[18] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," in *Proceedings of USENIX NSDI*, 2004.

[19] GreenOrbs. [Online]. Available: http://www.greenorbs.org

[20] L. Mo, Y. He, Y. Liu, J. Zhao, S. Tang, X.-Y. Li, and G. Dai, "Canopy Closure Estimates with GreenOrbs: Sustainable Sensing in the Forest," in *Proceedings of ACM SenSys*, 2009.

[21] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks," in *Proceedings of ACM SenSys*, 2006.

[22] MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller. [Online]. Available: http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf

[23] J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann, 2000.

[24] TinyOS TEP133. [Online]. Available: http://www.tinyos.net/tinyos-2.1.0/doc/html/tep133.html

[25] S. S. Kulkarni and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," in *Proceedings of IEEE ICDCS*, 2005.

[26] V. Naik, A. Arora, P. Sinha, and H. Zhang, "Sprinkler: A Reliable and Energy Efficient Data Dissemination Service for Wireless Embedded Devices," in *Proceedings of IEEE RTSS*, 2005.

[27] L. Huang and S. Setia, "CORD: Energy-efficient Reliable Bulk Data Dissemination in Sensor Networks," in *Proceedings of IEEE INFOCOM*, 2008.

[28] I.-H. Hou, Y.-E. Tsai, T. F. Abdelzaher, and I. Gupta, "AdapCode: Adaptive Network Coding for Code Updates in Wireless Sensor Networks," in *Proceedings of IEEE INFOCOM*, 2008.

[29] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *Proceedings of ACM ASPLOS*, 2002.

[30] J. Koshy and R. Pandey, "VM*: Synthesizing Scalable Runtime Environments for Sensor Networks," in *Proceedings of ACM SenSys*, 2005.

[31] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A Dynamic Operating System for Sensor Nodes," in *Proceedings of ACM MobiSys*, 2005.

[32] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of EmNets*, 2004.

[33] J. Koshy and R. Pandey, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," in *Proceedings of EWSN*, 2005.

[34] J. Jeong and D. Culler, "Incremental Network Programming for Wireless Sensors," in *Proceedings of IEEE SECON*, 2004.

[35] P. von Richenbash and R. Wattenhofer, "Decoding Code on a Sensor Node," in *Proceedings of IEEE DCOSS*, 2008.

[36] J. Hu, C. J. Xue, and Y. He, "Reprogramming with Minimal Transferred Data on Wireless Sensor Network," in *Proceedings of IEEE MASS*, 2009.

[37] N. Tsiftes, A. Dunkels, and T. Voigt, "Efficient Sensor Network Reprogramming through Compression of Executable Modules," in *Proceedings of IEEE SECON*, 2008.

[38] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks using Random Linear Codes," in *Proceedings of ACM/IEEE IPSN*, 2008.

[39] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. F. H. III, and M. Zorzi, "SYNAPSE: A Network Reprogramming Protocol for Wireless Sensor Networks using Fountain Codes," in *Proceedings of IEEE SECON*, 2008.

[40] K. Lin and P. Levis, "Data Discovery and Dissemination with DIP," in *Proceedings of ACM/IEEE IPSN*, 2008.

[41] T. Dang, N. Bulusu, W. chi Feng, and S. Park, "DHV: A Code Consistency Maintenance Protocol for Multi-hop Wireless Sensor Networks," in *Proceedings of EWSN*, 2009.

[42] P. Dutta, J. Taneja, J. Jeong, X. Jiang, and D. Culler, "A Building Block Approach to Sensornet Systems," in *Proceedings of ACM SenSys*, 2008.

[43] ST M25P80 Datasheet. [Online]. Available: http://www.datasheetcatalog.org/datasheet/stmicroelectronics/8495.pdf

[44] AT45DB041B Datasheet. [Online]. Available: http://www.atmel.com/dyn/resources/prod_documents/doc3443.pdf

[45] Q. Cao, T. Abdelzaher, J. Stankovic, and L. Luo, "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks," in *Proceedings of ACM SenSys*, 2008.