Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks

Joel Koshy
koshy@cs.ucdavis.edu

Raju Pandey
pandey@cs.ucdavis.edu

Department of Computer Science
University of California, Davis
Davis, California 95616, USA

Abstract

With sensor networks expected to be deployed for long periods of time, the ability to reprogram them remotely is necessary for providing new services, fixing bugs, and enhancing applications and system software. Given the envisioned scales of future sensor network deployments, their restricted accessibility, and the limited energy and computing resources of sensors, transmitting raw binary images is inefficient. We present a technique to minimize the cost of application evolution by remotely and incrementally linking updated modules at the base station, and distributing deltas of the pre-linked software modules. This paper provides details of our implementation, some preliminary results, and surveys critical research issues in developing a comprehensive framework for reprogramming sensor networks.

1. INTRODUCTION

Wireless sensor networks (WSNs) are large networks of deeply embedded devices that coordinate to perform continuous sensing tasks over large spatial and temporal scales [1]. By providing fine-grained and unintrusive monitoring in real-time, WSNs allow tight integration of the physical world with a computing system infrastructure. WSNs have been demonstrated to play an important role in a rich variety of applications such as environmental and civilian infrastructure monitoring [6], wildlife tracking [19], military surveillance, etc.

Many of these applications have reprogrammability needs ranging from parameter changes for fine-tuning applications, to whole system reprogramming[1](*reflashing*). Depending on the application, updates may be pushed by the programmer to the network, or pulled by self-tuning applications. Given the inaccessibility of sensing devices due to large scale deployment in physical settings that may be impossible to reach, it is critical to anticipate strategies for software evolution in the early stages of system design.

The ability to remotely reprogram sensor nodes is necessary for several reasons. First, the scale and distributed nature of WSN applications makes it difficult to get things right the first time. Bug fixes, run-time application adaptation and other software maintenance concerns can be addressed only by employing a reprogramming mechanism. Second, application specialization to optimize energy usage and performance for network longevity may be possible only at run-time. WSNs tend to be highly sensitive computing environments in which small local changes can effect significant global consequences. Normally, an ideal configuration can only be attained empirically at run-time, because it is not possible to anticipate every application scenario. For example, it may be necessary to choose a routing protocol from a protocol suite at run-time, suited to prevalent conditions [14]. Similarly, knobs such as power management, radio frequency modulation and dynamic voltage scaling algorithms may need to be adjusted at run-time. Third, it is likely that some WSNs will be deployed for long periods of time and provide different services at different times. Due to storage constraints, it is infeasible to load all these services into the nodes prior to deployment. Instead, applications and services could be swapped in and out depending on contexts of use such as current time, location, user input, and environmental stimuli. Run-time service composition is also useful in context-aware pervasive computing environments with small computing devices [3].

Reprogramming mechanisms should satisfy the following constraints:

1. *Unintrusiveness:* WSNs are tightly coupled to their

1 Whole system reprogramming is the process of erasing the program memory and uploading a new binary image.

surrounding environment and are often physically inaccessible once deployed. Reprogramming should be possible without physical access to the nodes, and wireless transmission of updates is the only acceptable solution.

2. *Low overhead:* To facilitate dense deployments, nodes are designed to be inexpensive and thus limited in their energy supply, computing resources and form factor [15]. Mechanisms for reprogramming should be efficient so as to not stray from the original objective of long-term sensing. Also, reprogramming has to take place fairly quickly. Applications may have real-time constraints, and a lengthy reprogramming phase can violate the continuous monitoring requirement of some WSNs. Consequently, communication and program memory rewriting should be minimized as far as possible, as these are the main bottlenecks.

3. *Resource awareness:* Node-resident software necessary for reprogramming should not be compute intensive, and should have low memory requirements so as to reserve sufficient space for the system software and application itself.

A comprehensive framework for reprogramming WSNs would contain several components. A *builder* at the base station is responsible for producing updates, and an *injector* injects them into the network. A *code distribution protocol* (CDP) [33, 25, 17, 22] propagates the updates throughout the network. In addition to being resource-efficient, the CDP should guarantee that all intended targets receive updates in their entirety, and with low latency. During code distribution, nodes should be able to synchronize with each other to avoid concurrent execution of incompatible versions of code. At the recipient nodes, several services may be needed, such as an *authenticator* to prevent malicious updates, a *checker* to check program integrity and correctness, a *bootloader* to rewrite the program memory, and a *restorer* to trigger recovery mechanisms in the event of a failed update. At a higher level, a *distributed version control system* will be needed to maintain version trees, to facilitate checkpointing and reverting to stable versions when necessary. Of these components, the builder, the injector, the CDP and the bootloader are necessary. The other components may be present according to available resources and level of sophistication needed.

In this paper we present a novel technique for updating program binaries on sensor nodes. We focus mainly on the builder and bootloader components. The technique is based on two key ideas. First, the size of the updates can be reduced if changes in applications are identified at the application level. This allows us to precisely capture direct changes, as opposed to indirect changes that occur due to semantic dependencies among program elements. This significantly reduces the size of the updates that must be distributed in the network. Second, the task of updating and modifying binaries can be partitioned between a base station and sensor nodes, thereby reducing the computing and space overhead at the sensor nodes.

We have implemented the technique for the Mica family of sensor nodes [9]. Our preliminary results indicate that the approach outperforms both reflashing and diff-based approaches significantly. Update sizes are reduced to 1.38%–56.61% of updates generated by diff-based approaches for incremental changes. Section 2 contains an overview of existing approaches to reprogramming WSNs. We describe our methodology in Section 3. Section 4 gives details of our implementation, and preliminary results are provided in Section 5. We discuss a number of open issues and tradeoffs in building a reprogramming framework in Section 6, and conclude in Section 7.

2. RELATED WORK

Mainstream software systems have repeatedly proven the need for adaptivity and extensibility. While the nature and frequency of changes may differ in WSNs, software artifacts will change after they have been deployed in the field, and a careless approach to sofware evolution will result in short-lived applications. In conventional systems, software maintenance accounts for 60-70% of software cost [12]. About 50% of this effort is perfective, 21% corrective, 25% adaptive, and 4% preventive [26]. Thus, a variety of workarounds such as wrappers and patches have been used to enable modification [11]. These solutions are unlikely to work in the WSN domain, as they usually result in software that is bloated, fragile, and bug-ridden. Various approaches have been proposed to address these problems.

In SensorWare [4], services are grouped into theme-related APIs with Tcl-based scripts as the glue. Scripts located at various nodes use these services, and collaborate with each other to orchestrate the dataflow to assemble custom networking and signal processing behavior. Application evolution is facilitated through editing scripts and injecting them into the network. Maté [24] is a compact virtual machine designed specifically for WSNs, built over TinyOS [16] — a component-based operating system for sensor nodes. Maté contains intrinsic support for application code distribution and code updating. Although virtual machines are promising as system software for WSNs, their space and energy overheads can render them counterproductive. Both SensorWare and Maté are limited in that they support application updates only (by replacing high-level scripts), and do not permit the lower level binary code (e.g., system software) to be modified. Our approach to reprogramming allows arbitrary changes to code, including sys-

tem software and, thus, can complement middleware techniques.

Reijers and Langendoen [32] use a *diff*-based updating scheme for WSNs. Updates are performed by rebuilding the modified application/system software at the base station, and generating a diff of the modified executable with the original. The diff is used to build a program memory edit script which is then propagated throughout the network. The size of the edit script is reduced through various optimizations, some of which are architecture-dependent. Jong and Culler [18] describe a similar approach using the Rsync algorithm [35]. A drawback of these approaches is that updates are essentially stateless, as the diff algorithm is unaware of the application structure. The sizes of edit scripts are not necessarily congruent to the extent of adaptation, because even small changes can result in code shifts that necessitate fixing up several branch targets (jumps and function calls). Thus, the scheme does not always scale with the size of the diff, and beyond a certain point, even reflashing may be preferable. However, these studies confirm the benefits of using diff algorithms, and is the closest to our work in existing literature. In addition to generating diffs to encode changes to program memory, we reduce the effects of code shift by containing changes to functions within a *slop* region, as far as possible.

Impala [27] is a layered middleware architecture that enables application modularity, adaptivity and repairability. Its Application Updater allows software updates to be performed by linking in updated modules. However, the linking capability is limited, and updates are coarse-grained as cross-references between modules are not allowed. The dynamically linked functions are invoked indirectly through a table of function pointers, inducing a performance overhead. Also, Impala is targeted for unconventional nodes with considerable computing resources.

The TinyOS distribution includes support for single-hop over-the-air reprogramming (XNP) [10] on Mica motes. XNP and multi-hop reprogramming schemes such as MOAP [33] rebuild the modified application, and transmit the entire image. This does not scale to large networks due to the energy overhead associated with transmitting images. There can be significant latency in reprogramming on the Mica platform, because the entire image has to be downloaded via a low bandwidth radio link into the slower external flash, prior to actual update. Also, erasing and rewriting the entire program memory is slow and energy-intensive.

3. METHODOLOGY

3.1. Overview

To reduce the overhead in sending binary images, only diff-like updates are distributed. These updates are encoded into *diff-scripts* which are injected into the network. The bootloader applies patches by executing the scripts. This approach also exploits the fact that typical changes to running WSN applications are likely to be small.

Techniques to generate diff-scripts can differ in their awareness of the high-level aspects of the code being updated. In pure diff-based techniques [32], modified and unmodified objects are relinked together as is done normally, if the application were being rebuilt. Diff-scripts are generated by feeding the original and rebuilt binary images as input to binary diff algorithms (e.g., suff [30] or bdiff [34]). A drawback of this approach is that diff-based algorithms operate at the byte level, and are not concerned with the application structure. This has some undesirable effects that we consider below.

In our approach, we modify the object linking procedure itself to facilitate the generation of more concise diff-scripts called *deltas*. Thus, the application's structural evolution is not detached from the patching mechanism. Rather, it drives the generation of deltas. The linking approach better conforms to the intuition that reprogramming is essentially relinking and replacing modified modules. It retains knowledge of program structure, and can enforce a certain discipline in software evolution.

Ignoring the program structure can have an adverse impact on the cost of program memory rewriting at the node, and can preclude various optimizations that we discuss later. The program memory contained in most microcontrollers in current use in the WSN domain is realized through flash memory technology. This is the newer generation of flash memory that provide self-programming capability. Although flash memory has its advantages, reprogramming is not straightforward, and can only take place at the granularity of a page. Thus, even if only a few bytes in a page need to be altered, the entire page is buffered in SRAM and modified. The flash page is erased, and the modified page is transferred from SRAM to flash. In addition to the latency in copying data to and from the buffer, the actual erase and rewrite operations are slow and power hungry. Another potential problem is that flash pages have a cycle limit for erasing and rewriting. Ideally, if an application requires self-programming, it should enforce *cycle-leveling*[2]. Table 1 highlights the relevant properties of a typical flash memory unit contained in the Atmel ATMega128 microcontroller [2].

2 Cycle-leveling (or wear-leveling) is the technical term for wearing down all flash pages as evenly as possible.

Memory Size	128KB
Page Size	256 bytes
Read Latency	3 CPU cycles/byte
Write/Erase Latency	3.7–4.5 ms/page
Write/Erase Cycle Limit	10,000
Programming Current	2–3 mA at V_{cc} = 3 V 5–7 mA at V_{cc} = 5V

Table 1. ATMega128 Internal Flash Memory Characteristics.

Thus, an efficient reprogramming system should not only reduce communication overhead by transmitting incremental patches, but it should also ensure that the amount of flash rewriting at the recipient is minimized. While diff-scripts generated by diff-based approaches reduce transmission overhead, they cannot by themselves guarantee low overhead in rewriting the flash. This is a direct consequence of being detached from the higher level aspects of application evolution.

Consider a set of functions `f`, `g`, and `h` laid out in flash as shown in Figure 1. Function `f` invokes `g` and `h`. If `g` is updated, there are two interesting possibilities — growing and shrinking. When `g` shrinks, all the code below it is shifted to lower addresses. Thus, even though `h` (and other functions below it) do not change in their content, they need to be shifted to their new locations. If there are any calls to these functions (e.g., the call to `h` from `f`), the targets of those call instructions need to be patched by re-applying the corresponding relocation entries[3]. Basic diff-scripts contain `copy` and `insert` (or `add`) instructions [5]. Copy instructions specify a region of the original, that can be copied to the modified version. Insert instructions are used to introduce the actual changes between the two versions, and are therefore larger than copy instructions.

In this example, `copy` instructions could be used to slide code to lower addresses (starting with function `h`). However, `insert`'s will be required for patching calls to any function that follows `g` (e.g., the call to `h` from `f`), as these functions are relocated. Updating `g`'s implementation will also require `insert`'s (and possibly a few `copy`'s). The bootloader will have to rewrite pages 15 onwards, due to code shift. Page 14 also needs to be rewritten to update the target of the call to `h`.

Expanding `g` has similar repercussions. If `g` remains the same size, diff instructions are generated only to modify the region it occupies. Copy instructions will be generated for the rest of the flash. We notice that although the diff-scripts can be small, several pages of flash need to be rewritten.

3 Relocation in linking teminology, is the process of binding references to absolute addresses.

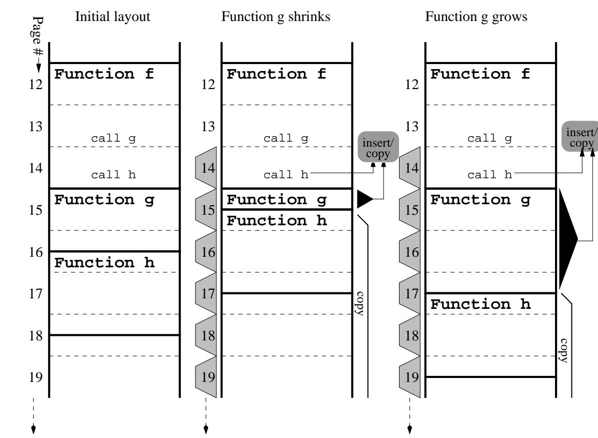

Figure 1. Pure diff-based approach. Tabs indicate which pages are rewritten.

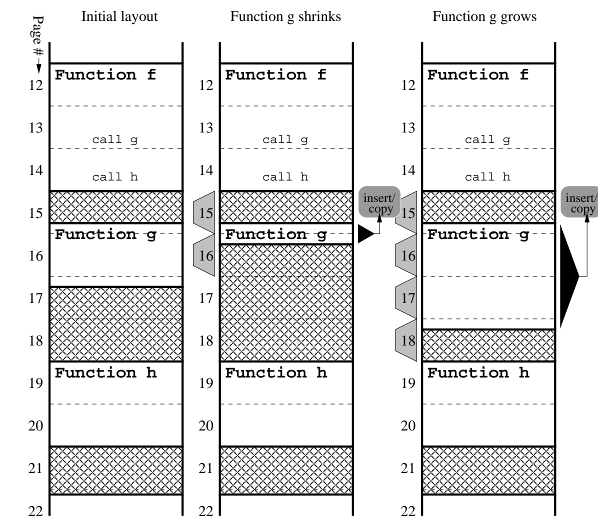

Figure 2. Incremental linking approach. Slop space is hatched, and tabs indicate which pages are rewritten.

In other words, the sizes of the diff-scripts are inconsistent with the extent of actual adaptation.

Our approach is to modify the linking procedure, to function in an incremental fashion. By doing so, we minimize the costs of transmission and flash rewriting. Flash rewriting is reduced by ensuring that as far as possible, functions that do not change are not shifted. Additionally, functions are provided with a *slop* region [31] to grow without running into another function. If a function attempts to grow beyond its allocated slop region, it is relocated to a larger region, with additional slop space.

To illustrate the incremental linking approach, consider the earlier example. The linker is modified to provide slop space for each function. Figure 2 shows the effect of `g` shrinking and growing by small amounts. The slop provides space for `g` to grow, and no code needs to be relocated. Thus, no call targets need to be updated. The only pages that need to be rewritten are due to the `insert` 's and `copy` 's necessary to update `g`.

In addition to reducing flash rewriting, the delta is smaller, and the memory required for rebuilding the flash memory is less. Figure 1 does not show this detail, but in current diff-based approaches, `insert` instructions are used to edit the original image or add new data, and `copy` 's are used to copy *all* the unmodified regions. Thus, the entire flash is rebuilt in a temporary buffer (such as the EEPROM or external flash), and modified pages are rewritten. Some diff formats provide a windowing mechanism [21], which allows for a smaller buffer by working with small segments of the original. In our incremental linking approach, diff instructions are generated only for the modified pages, and only those pages need to be buffered. Utilizing page-sized windows can reduce or eliminate external flash memory requirements by working with smaller buffers in SRAM.

3.2. Linkers and loaders

Although hardware features and language related idiosyncracies make linkers complex programs, in principle, a linker's task is simple — it binds abstract names to concrete values [23]. A linker combines object modules (relocatable object files) and run-time libraries into a single load module. It assigns addresses to symbols, resolves cross-references between modules, and lays out sections. Laying out sections can necessitate the relocation of input sections. Linkers typically maintain large data structures to store information such as symbols exported by input objects, relocation entries, etc. Linkers function mainly at compile-time, although some linker related activities (including dynamic linking) take place at run-time.

Loaders are primarily responsible for loading the program into main memory for execution. Loaders can be extremely complex if special hardware features are needed. However, most sensor nodes do not use such features, and programs are typically loaded once into flash, prior to deployment. A more capable microcontroller with secondary storage would require run-time loading (and sometimes relocation) to load the program from secondary storage into main memory. The boundary between linkers and loaders is fuzzy, and their activities sometimes overlap.

3.3. Remote incremental linking

An incremental linker relinks only those modules that have changed since the last pass of linking. Therefore, it needs to maintain additional information about the input objects and their constituent program elements. While the traditional objective of incremental linking is to reduce the edit-recompile-relink cycle latency to improve developer efficiency [31, 29], our goal is to reduce transmission and flash rewriting overheads.

Using a linking mechanism raises conflicts with the requirements outlined earlier. Resource awareness can be violated, because the linker can require considerable computing resources and memory. Linking requires symbol tables and other data structures which can be too large for resource-constrained nodes[4]. The low overhead requirement can also be compromised because wireless transmission of symbol tables necessary for linking and the object files themselves, will be slow and energy-intensive. We approach this problem by linking remotely at a more capable device such as the base station, and generating small deltas which are then distributed.

4. DESIGN AND IMPLEMENTATION

We have extended the AVR port of the GNU linker, to perform incremental linking. This requires the use of the Binary File Descriptor Library (*libbfd*) [7] to manipulate object files in a generic fashion. *libbfd* is useful for working with various binary formats, and provides suites of routines for performing linking related activities such as relocating sections and maintaining hash tables for symbol resolution. It also makes porting to other targets easier, by working with objects using an internal canonical representation, and writing out processed objects in standard formats using suitable backends.

Our target platform is the Mica2 *mote*, originally developed by the UC Berkeley research group [9]. Applications are written in C, and ELF object files are built using the AVR port of the standard GNU binary utilities. Linking takes place at the base station, which records information necessary for future incremental links in local data structures. The program image is then flashed onto the device. At run-time, application updates trigger incremental linking. Deltas are generated, and transmitted to the device through the wireless or serial link. Before describing the linker in detail, we highlight the hardware features that affect the linker's functioning.

4 Symbol tables and other auxiliary data structures of typical object files occupy 45–55 percent of the file.

4.1. Mica2 summary

Berkeley motes are popular sensor nodes, in use by over 100 research organizations [15]. We use the Mica2 for our work, but the implementation is portable across the Mica family. These motes are constructed from off-the-shelf components. The central microcontroller unit (MCU) is the Atmel ATMega128 [2], which is responsible for computation, and coordinating sensing and communication. A 512KB external flash unit (AT4DB041B) interacts with the MCU through the serial interface, and is intended to store sensor data, and program images during reprogramming.

Atmel ATMega128: The ATMega128 contains 128KB of self-programmable program memory (flash), 4KB of internal SRAM, and 4KB of internal EEPROM. Reprogramming is complicated by the Harvard architecture and the inability to execute machine instructions from SRAM. The flash memory characteristics were given in Table 1. Figure 3 shows the layout of flash (as we have configured the MCU) and SRAM memories. Flash is separated into two main sections — the application section and the bootloader section (BLS). Their relative sizes can be configured by fuse bits on the MCU. Also, the two sections can have different levels of protection depending on the setting of certain lock bits. The `spm` instruction, which rewrites flash pages, is enabled only in the BLS. Using this instruction, a bootloader can modify any part of the flash, including itself. Flash is also separated into the Read-While-Write (RWW) and No-Read-While-Write (NRWW) sections. This allows the flash to provide true *read-while-write* operation. As long as the RWW section is being reprogrammed, the NRWW section can be read. Any attempt to read code in the RWW section during reprogramming (e.g., by a `call`/`jmp`/`lpm`) might result in an unknown state. If the NRWW section is being reprogrammed, the CPU is halted until reprogramming is complete.

By locating less volatile critical code in the NRWW section, and more volatile application code in the RWW section, it is possible to allow critical functions to continue during a lengthy reprogramming phase. This is why we have to shift the interrupt vector table to the NRWW section. The default location (`0x0000` – `0x0044`) would necessitate disabling interrupts during reprogramming. The data initialization region is described in Section 4.2.

The first 256 bytes of SRAM are reserved for system registers such as I/O registers, general purpose registers. The layout of `.data`, `.bss`, the heap and the stack is similar to conventional SRAM organization. To deal with updates, there are differences in how variables are laid out within the `.data` and `.bss` sections. This is discussed in greater detail in Section 4.2.

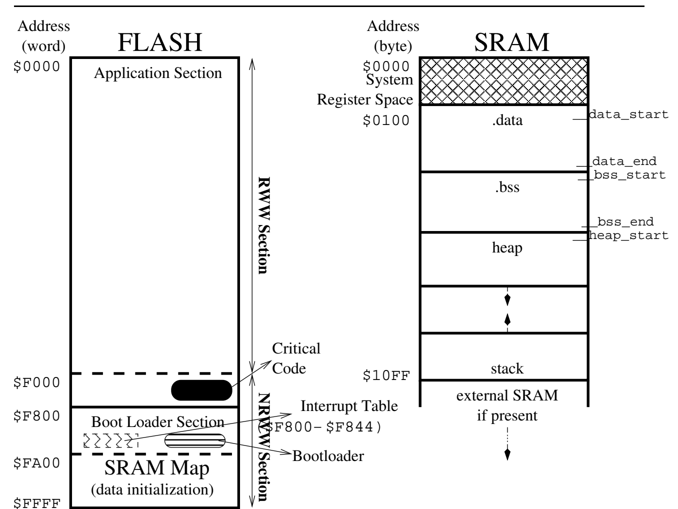

Figure 3. ATMega128 memories.

Boot process: When a fully linked executable is programmed onto the device, values of initialized data included at the end of the object are also stored in flash. When the MCU is reset, initialization code sets up the stack and copies these values to the run-time address of the corresponding variables in the `.data` section in SRAM. After this, the `.bss` section is cleared, and control jumps to the application's function `main`. The data initialization region normally follows the `.text` segment. However, when we update a module or add a new one and possibly introduce new initialized data, their values need to be appended to the data initialization region. Unless there is enough slop space, this can cause subsequent code in the `.text` segment to shift. Therefore, we moved this region to the end of flash and modified the initialization code accordingly.

4.2. Implementation details

Remote incremental linker

To provide remote linking services, the base station needs to manipulate and link together relocatable ELF objects [8]. From a linker's perspective, an ELF file consists of a set of logical sections described by a section header table. Corresponding sections from various objects in the link are laid out consecutively in an output segment in the load module. Thus, the final executable consists of a set of segments which are composed of processed sections pulled from various objects. Typical ELF sections include `.text`, `.data`, `.bss`, `.symtab` (symbol table), `.rel*` sections containing relocation entries, and sections containing metadata useful for debugging. The base station also needs to record aux-

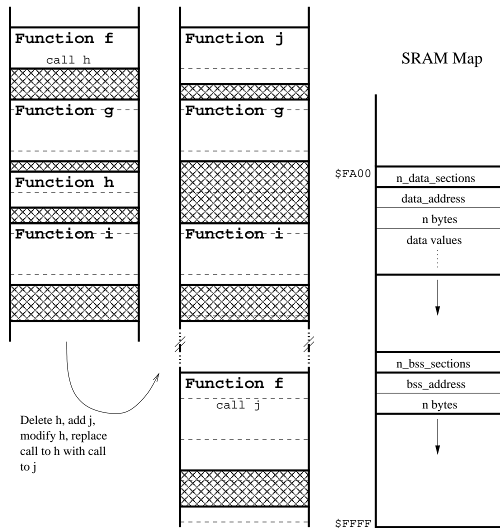

Figure 4. Updating flash; data initialization region layout.

iliary information. It maintains a map of the program and data memories of the nodes in order to allocate space for incrementally added entitites. It also keeps its own symbol table for resolving references from modules that may added in the future. Symbol table entries include information such as size, run-time addresses, function and variable dependencies, and slop space allocated to functions.

In our implementation, updates are pushed by the application programmer after changes are made to the application. Changes could introduce new symbols, and modify or delete existing ones. We now describe the process of updating sections in flash and SRAM.

Flash: Modifying symbols in the .text region (e.g., functions, constant data) is complicated by the intricacies of flash rewriting. Consider the example given in Figure 4, which outlines some possible modifications. A node is initially deployed containing a module M that includes functions f, g, h and i. During the initial link, dependencies (e.g., f calls h) and other metadata such as function size, slop space, symbol addresses, etc., are recorded. At run-time, M is altered by deleting function h, and replacing it with the global function j. When h is deleted, the incremental linker analyzes its dependencies. Functions that depend on h become inconsistent, and relocation entries corresponding to h (i.e., references to h) are deleted from the dependency data structure. In this example, to preserve consistency, function f is modified, and it calls the newly introduced function j.

The incremental linker implements a memory manager that allocates and deallocates flash for functions. The memory manager also manages allocation and deallocation of SRAM to variables. The general sequence is to have the linker's memory manager layout the .text and .data segments, and then perform relocation to resolve references. The space occupied by h is freed to the manager. Function h is relatively small, and g's slop space is limited. In this case, the manager allocates this space to g's slop. Alternatively, the space can be reserved for use by a function that may be added in the future.

In this particular example, f grows beyond its slop space. When a function grows beyond its slop space, various options are evaluated. One option is to shift function f to a fresh region of flash with additional slop. Another is to attempt to shift one or more functions that follow (g and h) into their slop regions, so that f's slop space is increased. Doing so may be helpful if there are more references to the modified function than to the functions that follow. The linker selects the strategy that minimizes the number of pages to be rewritten.

In this example, the linker places j at f's old location. After finalizing the locations of symbols, it reapplies relocations corresponding to functions that have shifted. For example, all calls to f are updated with its new address. Function f itself contains relocation entries such as the reference to j that need to be applied. The newly introduced function j may contain references to external symbols such as g or variables in .data . The linker applies relocation entries corresponding to these references in the traditional manner. When updating interrupt handlers, the interrupt vector table may need to be updated as well. If the interrupt handler shrinks or remains the same size, then a routine function update can be carried out. However, if it grows beyond its slop space and is relocated, its address in the interrupt vector table needs to be fixed. Interrupts are disabled during this operation, since the table resides in the NRWW section.

SRAM: Modules with updated functions may alter the variables that they define. Variables could be deleted, overridden, or newly introduced. .data variables are explicitly initialized by the programmer, while .bss variables are cleared to zero. Conventional linkers store the initial values of .data variables after the .text section in flash, and initialization code (__do_copy_data) copies these values to the actual locations in SRAM, and clears out .bss (__do_clear_bss). This approach can be problematic because if updates add new code, they will be placed immediately after the data initialization section. Further updates that introduce new initialized variables will necessitate shifting that code. Thus, we have modified the conventional approach to accomodate the possibility of updates.

The last 3KB of flash is used as a memory map of the SRAM. With this space we cannot map the entire 4KB of SRAM, but it is sufficient to map .data and .bss sections of typical proportions. It is also possible to use the

4K internal EEPROM or a portion of the external flash to map the SRAM. Figure 4 shows the general layout of the data initialization region. The first byte indicates the number of data sections that need to be initialized. Each subsequent section is specified by the address in SRAM, the number of bytes, and the data values. After reset, the initialization code iterates through these sections, and copies the data to the specified addresses. The formatting of the `.bss` section initializers are similar, but the data values are not included since `.bss` variables are cleared to zero.

When a variable is deleted, the space it occupies in SRAM is freed to the memory manager. The data initialization region does not need to be changed. The earlier value is still copied by the initialization routine, but is not used by the application. If a module introduces new variables, the memory manager allocates space in SRAM (in `.data` or `.bss` depending on the variable definitions), and their initial values are inserted into the data initialization region of flash. If a variable definition is overridden, new space needs to be allocated if the size of the new definition is larger than the old definition. Otherwise, the new definition can simply be placed at the location of the earlier definition, and the excess space is freed to the memory manager. The data initialization table in flash is edited to incorporate the new value. When the bootloader determines that the data initialization table needs editing, it disables interrupts as necessitated by any update to the NRWW section.

Delta generation

After linking, we feed the original flash image and pages of the rebuilt image, as input to a binary diff algorithm. We use the Xdelta algorithm [28] to generate concise diff-scripts called deltas. The target window size is set to the page size, so that deltas are generated only for pages that are modified. This also allows updates to be applied at the granularity of a page, reducing the amount of temporary storage needed to store images before they can be applied. This is a significant advantage over other schemes [18, 32] that require upto twice the amount of program memory in secondary storage to rebuild the entire flash image before writing it to the program memory.

The delta is encoded using the VCDIFF format [21]. The VCDIFF format allows the use of any diff algorithm, encodes diffs compactly in a portable manner, and is designed for efficient decoding. In addition to `copy` and `add` instructions, it provides a `run` instruction to encode a continuous run of a byte, an opcode table to compactly encode frequently occuring pairs of `copy` 's and `add` 's, and address caches to reduce the length of addresses that need to be encoded for `copy` instructions. Xdelta supports other features that we disable. For example, the resulting delta can be compressed further by a secondary compressor (e.g., bzip2).

Bootloader

When the node receives a delta, control is transferred from the application to the bootloader. By using the read-while-write capability, it is possible to pipeline communication and reprogramming. When a delta is received, it is loaded into a buffer. The delta is then interpreted, and the updated page is rebuilt in a second buffer. After the page is rebuilt, the bootloader writes the page back to internal flash. Rewriting a page involves considerable latency. By locating code implementing communication in the NRWW section, the next delta can be downloaded and interpreted in the first buffer, while the previous page is being written to flash. After all the deltas are received and applied, the bootloader resets the MCU.

Memory management

Our current implementation provides a simple memory manager with a fixed allocation and deallocation policy. Slop space is allocated in linear proportion to the size of functions. Ideally, slop space should be allocated according to a function's volatility [31]. This could be determined through profiling changes to the application, pragmas supplied by the programmer, or a combination of these approaches. Providing too much slop will waste memory, while too little slop can cause frequent relocations.

The manager attempts to service flash allocation requests such that all regions of flash are used. To do this, it keeps track of flash usage with the help of a circular buffer, and unidirectionally traverses the buffer when requests are serviced. Deallocated chunks are returned to the circular buffer.

After several updates, flash and SRAM memory may become fragmented, and it may be beneficial to compact discontiguous regions. Compaction is expensive because of the large amount of copying and rewriting necessary. The task is complicated even further because shifting code or data segments around requires reprocessing of relocation entries — if these references change, the pages they are contained in need to be rewritten. At a certain stage, falling back to a one time reflashing, or rebuilding the application and applying a diff to the entire flash memory may be beneficial. Currently, we do not do any compaction, as this becomes an issue only after several updates are performed.

5. EXPERIMENTAL EVALUATION

In this section, we provide preliminary results of using our incremental linker in tandem with the Xdelta algorithm.

We compared the performance of incremental linking with a pure diff approach and with whole system reprogramming (WSP), for cases ranging from changing con-

Expt.	Application	Initial binary size (bytes)	Final binary size (bytes)	New functions	Modified functions	Deleted functions
I	Blink to Blink (change constant)	5688	5688	0	1	0
II	CntToLedsAndRfm to CntToRfm	28268	27092	0	3	24
III	CntToLedsAndRfm to CntToLeds	28268	6600	0	4	198
IV	CntToRfm to CntToLedsAndRfm	27092	28268	24	4	0
V	Blink to CntToLedsAndRfm	5688	28268	221	4	6
VI	CntToLedsAndRfm to Blink	28268	5688	6	3	221
VII	TOSBase to SecureTOSBase	26730	46420	79	9	4

Table 2. Upgrade scenarios.

Expt.	WSP	Pure Diff (Xdelta)					Remote Incremental Linking						
	File (bytes)	Delta (bytes)	cpy	add	run	Super ops.	Delta (bytes)	Delta % Diff	cpy	add	run	Super opcodes	Pages rewritten
I	5688	27	2	1	0	0	27	100.00	2	1	0	0	1
II	27092	6370	760	430	1	153	88	1.38	6	3	0	0	3
III	6600	2398	249	151	1	56	145	6.05	14	9	0	2	5
IV	28268	6639	826	452	0	165	678	10.21	105	47	7	19	10
V	28268	11839	1376	694	2	341	10846	91.61	1784	710	30	273	128
VI	5688	2365	246	148	1	42	264	11.16	28	17	4	1	7
VII	46420	16882	2186	1018	0	549	9557	56.61	1444	611	14	299	121

Table 3. Delta sizes with incremental linking, pure diff and WSP.

stants, to major upgrades. Table 2 summarizes the upgrade scenarios we considered. All the applications were taken from the examples in the TinyOS distribution. The initial and final binary sizes were obtained by compiling the applications with compiler optimizations turned off, and without any function inlining. The C files generated by the *nesC* compiler [13] were modified directly, to create patches between applications. The original applications and the patches were then incrementally linked, and deltas were generated. Blink to Blink is the simplest experiment which involves changing the blink rate constant. The application pairs considered in experiments II, III, IV and VI required minor changes at the application structure level, adding or modifying only a few functions. Blink to CntToLedsAndRfm is a major upgrade which involves the addition of 221 new functions and modifies 4 existing functions in Blink. SecureTOSBase is a TinySec-aware [20] TOSBase, and is a more practical example of an incremental change. TinySec provides cryptographic functions, and makes changes to the TinyOS radio stack to redirect byte level radio events to the TinySecM module.

Table 3 summarizes the characteristics of the deltas produced. All the experiments, with the exception of experiment I, have to deal with code shifts that result in large delta sizes for the pure diff and WSP approaches. In the absence of code shift, the incremental linking and pure diff approaches yield identical deltas. However, when there is code shift during an incremental or a major upgrade, the pure diff approach performs poorly. Deltas produced by incremental linking range from 1.38%–56.61% of the size of deltas produced by the pure diff approach for incremental changes — that is, when functionality is added to or removed from an existing application. For a major upgrade (experiment V), incremental linking produces a delta that is 91.61% of the size of the corresponding pure diff delta.

Thus, by containing code shifts within slop spaces as far as possible, incremental linking yields significantly more concise deltas. The smaller deltas enable code distribution to take place rapidly and more efficiently, reduces the number of pages that need to be rewritten, and reduces the memory resources required at the recipient nodes.

6. DISCUSSION

Developing a suitable framework for reprogramming is a multifaceted undertaking, and we describe some optimizations and additional features that need to be considered. The purpose of this discussion is two-fold. First, it suggests qualitative measures for evaluating reprogramming techniques. Second, we hope to motivate much needed research along these directions.

Remote vs. local linking: Performing linking at the base station alone has some disadvantages. First, the linker needs to maintain state information for all nodes. Additionally, deltas that are transmitted may be node-specific since linking is done at the base station. Not all nodes are going to have identical memory maps and modules, leading to several patches sent across the network for each update. This can congest the network, and also lead to significant energy drain.

Shifting all the functionality of the linker to the sensor node is not practical. Relocatable object files with large symbol tables will need to be transmitted at a high cost to nodes, and stored in the limited memory available. Additionally, libraries for performing linking cannot be loaded into the nodes. The advantage in local linking is that the object files distributed by the base station will be more portable across the sensor devices, because symbols are not bound to addresses.

An intermediate solution would be to distribute partially linked modules, which resolve and discard symbols common to all nodes, but retain node-specific symbols. It may also be useful to employ a dedicated high-end node, to perform *proxy linking* for more limited nodes. Such nodes can receive a partially linked (possibly compressed) delta from the base station, decompress it if necessary, link it for a number of nodes in the neighborhood, and transmit node-specific fully linked patches locally.

Memory management: The memory manager is a critical component of the incremental linker, as it effectively drives the placement of newly introduced variables and functions, by servicing memory requests from the linking routines. Since the linker interacts considerably with both program and data memory management, some of its functionality has to be placed at the base station, to avoid overhead in communicating with a memory manager at the node. Still, a lightweight memory manager may need to communicate some node-specific state to the base station. The memory manager should: (i) minimize changes to internal flash, internal eeprom and external flash, as these modifications are slow and energy-intensive; (ii) allocate space and slop for symbols intelligently, according to their usage patterns; and (iii) use the flash uniformly for even wear.

Since flash rewriting is costly, frequently changing functions, or functions that are closely related should be colocated as far as possible. Our current allocation scheme attempts to service allocation requests across the entire range of flash, but does not guarantee cyclic-leveling. Cyclic-leveling and minimizing writes to flash are conflicting requirements, because our strategy in reducing flash rewriting is to avoid having to relocate functions that change frequently. Thus, those functions will be rewritten in place several times. However, the cyclic-leveling requirement is not as compelling as the need to reduce reprogramming, because most flash units can tolerate at least 10,000 erase/rewrite cycles per page. A sophisticated approach could utilize static analysis and/or run-time profiling to predict or report such usage patterns to the linker, so that it can lay out symbols and sections more optimally. Heuristics could be developed to anticipate the nature of updates for further efficiency. For example, the amount of slop space to allocate to a function depends on how likely the function is expected to change. Additional research is required to develop cost models for communication, computation and reprogramming flash memory. This could be used to analyze the effect of application behavior and model incremental linking as an optimization problem with minimizing the size of deltas, and reducing communication and flash rewriting as the objective functions. Sensitivity analysis could be applied to study the effects of memory allocation, slop management, etc., on linker performance.

Safety and reliability: Security and fault tolerance are critical, but difficult to provide in the presence of an update mechanism. An update facility can potentially open a door through which rogue code could be distributed to cripple an entire network. Thus, authentication schemes, and secure and reliable communication channels need to be in place to enable safe updates. Update protocols should be robust to localized failures or security breaches in the network. In general, security and fault tolerance in sensor networks should be based largely on the scale of the system and on redundancies that make the system reliable and safe. For example, privileges should be revoked from compromised nodes and transferred to trusted nodes, to contain any anomalies. Hardware features could be developed to protect against malicious updates. The ATMega128 for instance, has a minimal memory protection system. Lock bits can be configured in software to lock the memory to prevent reprogramming. However, with physical access to the devices, the serial or parallel programming interface could be used to unlock memory.

Software updates should also be checked for correctness. It is possible to enforce simple rules for maintaining consistency. In general, updating symbols should be reflected in changes in all references to those symbols. For example, if a function returning a byte is modified to return a word, the caller should deal with this change as well, to avoid stack corruption. Although detecting such inconsistencies is not always straightforward, it would be useful to trigger a roll back to a safe state, in case of incorrect or failed updates. By keeping a chain of deltas at the base station, reverse patches can be pushed to the network, and applied by nodes when needed.

Qualitative measures: In addition to the properties of unintrusiveness, low overhead and resource awareness, up-

date mechanisms can vary in *scope*, depending on what layers of the software architecture can be affected. Dynamically updating software components at higher layers (e.g., Maté bytecode) is in general easier than updating low level software which entails linking and binary editing. Update mechanisms can also vary in their *granularity*, depending on the unit of adaptation. Coarse–grained updates (eg. whole system reprogramming) are straightforward to implement, but expensive in terms of communication overhead and binary management at the source. More fine-grained updates (eg. procedure or statement level) are better suited for the WSN domain, but are more challenging to implement. Finally, updates may proceed at various levels of *interference*. A good update mechanism should minimize interference to the running application, and at the same time meet the intentions of the programmer. In Section 4, we described how critical tasks could continue even during the reprogramming phase. Depending on the scope of the update, it may be possible to simply suspend the application momentarily and resume execution at the end of reprogramming. For major upgrades, a reset is necessary. In some cases, if the context of execution and the update are mutually exclusive, execution could continue in parallel. To ensure that updates and application execution do not interfere, precautionary procedures such as stack analysis may be necessary, to ensure that functions or variables that are being updated are not active.

7. CONCLUSION

Incremental linking reduces the amount of code shift and thereby reduces the difference between successive versions of code. Generating deltas with the Xdelta algorithm and encoding them compactly with the VCDIFF format, further reduces transmission costs. The approach may be used with any differencing algorithm and encoding scheme, but those that belong to the copy/insert class of algorithms such as Xdelta and Rsync, are better suited for transmission than the insert/delete class of algorithms such as diff. While whole system programming may be justifiable for major upgrades, our approach addresses an important class of incremental updates typically observed after deploying WSN applications. Although our implementation is tailored to the platform we use, the methodology may be applied to most embedded platforms with similar constraints.

Acknowledgments

The authors would like to thank Earl Barr and Ingwar Wirjawan from the University of California, Davis, and anonymous referees for their insightful comments on an earlier draft of this paper.

References

[1] *Embedded, Everywhere*. National Research Council, 2001.

[2] Atmel Corporation. *ATMega128 Datasheet*.

[3] G. Banavar, J. Beck, E. Gluzberg, J. Munson, J. Sussman, and D. Zukowski. Challenges: An application model for pervasive computing. In *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking*, pages 266–274. ACM Press, Aug. 2000.

[4] A. Boulis, C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services*, May 2003.

[5] R. Burns. Differential compression: A generalized solution for binary files. Masters thesis, Department of Computer Science, University of California at Santa Cruz, Dec. 1996.

[6] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. *SIGCOMM Computer Communication Review*, 31(2 supplement):20–41, 2001.

[7] S. Chamberlain. *The Binary File Descriptor Library*, Apr. 1991.

[8] T. I. S. Committee. *Executable and Linking Format (ELF) Specifi cation*, May 1995.

[9] Crossbow Technology Inc. *Mica Motes*. http://www.xbow.com.

[10] Crossbow Technology Inc. *Mote In-Network Programming User Reference*, 2003. http://www.xbow.com.

[11] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, 2002.

[12] R. Fjeldstad and W. Hamlen. Application program maintenance study: Report to our correspondents. Tutorial on Software Maintenance. IEEE Computer Press Society, 1983.

[13] 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 the Conference on Programming Language Design and Implementation*, pages 1–11. ACM Press, June 2003.

[14] Y. He, C. Raghavendra, S. Berson, and B. Braden. A programmable routing framework for autonomic sensor networks. In *Proceedings of the Fifth International Workshop on Active Middleware Services*, pages 60–68, May 2003.

[15] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Communications of the ACM*, 47(6):41–46, 2004.

[16] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the Ninth Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Nov. 2000.

[17] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems*, pages 81–94. ACM Press, Nov. 2004.

[18] J. Jong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE International Conference on Sensor and Ad Hoc Communications and Networks*, Oct. 2004.

[19] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107. ACM Press, Oct. 2002.

[20] C. Karlof, N. Sastry, and D. Wagner. Tinysec: A link layer security architecture for wireless sensor networks. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems*, pages 162–175. ACM Press, Nov. 2004.

[21] D. Korn, J. MacDonald, J. Mogul, and K. Vo. RFC 3284: The vcdiff generic differencing and compression data format, June 2002.

[22] S. Kulkarni and L. Wang. Mnp: Multihop network reprogramming service for sensor networks. Technical Report MSU-CSE-04-19, Department of Computer Science, Michigan State University, East Lansing, Michigan, May 2004.

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

[24] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the Eleventh Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95. ACM Press, Oct. 2002.

[25] 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 the First Symposium on Networked Systems Design and Implementation*, pages 15–28, Mar. 2004.

[26] B. Lientz and E. Swanson. Problems in application software maintenance. *Communications of the ACM*, 24(11):763–769, 1981.

[27] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 107–118. ACM Press, June 2003.

[28] J. MacDonald. File system support for delta compression. Masters thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[29] D. Mikulin, M. Vijayasundaram, and L. Wong. Incremental linking on hp-ux. In *Proceedings of the Fifth USENIX Symposium on Operating Systems Design and Implementation*, pages 47–56, Oct. 2000.

[30] W. Obst. Delta technique and string-to-string correction. In *Proceedings of the First European Software Engineering Conference*, pages 64–68. Springer-Verlag, Sept. 1987.

[31] R. Quong and M. Linton. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems*, 13(1):1–20, 1991.

[32] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the Second ACM International Conference on Wireless Sensor Networks and Applications*, pages 60–67, Sept. 2003.

[33] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[34] W. Tichy. The string-to-string correction problem with block moves. *ACM Transactions on Computer Systems*, 2(4):309–321, 1984.

[35] A. Tridgell. Efficient algorithms for sorting and synchronization. Ph.D thesis, Australian National University, Feb. 1999.