Mitigating the Effects of Software Component Shifts for Incremental Reprogramming of Wireless Sensor Networks

Rajesh Krishna Panta, *Member, IEEE*, and Saurabh Bagchi, *Senior Member, IEEE Computer Society*

Abstract—Wireless reprogramming of sensor nodes is an essential requirement for long-lived networks because software functionality needs to be changed over time. The amount of information that needs to be wirelessly transmitted during reprogramming should be minimized to reduce reprogramming time and energy. In this paper, we present a multihop incremental reprogramming system called *Hermes* that transfers over the network the *delta* between the old and new software and lets the sensor nodes rebuild the new software using the received delta and the old software. It reduces the delta by using techniques to mitigate the effects of function and global variable shifts caused by the software modifications. Then it compares the binary images at the byte level with a method to create a small delta that needs to be sent over the wireless network to all the nodes. For the wide range of software change scenarios that we experimented with, we find that Hermes transfers up to 201 times less information than Deluge, the standard reprogramming system for TinyOS, and 64 times less than an existing incremental reprogramming system by Jeong and Culler.

Index Terms—Sensor networks, incremental reprogramming, deluge.

1 Introduction

Large scale sensor networks may be deployed for long periods of time during which the requirements from the network or the environment in which the nodes are deployed may change. This may necessitate modifying the executing application or retasking the existing application with different sets of parameters, which we will collectively refer to as *reprogramming*. Once deployed, it may be very difficult to manually reprogram the sensor nodes because of the scale (possibly hundreds of nodes) and the embedded nature of the deployment since the nodes may be located in places that are difficult to access physically. The most relevant form of reprogramming is *remote multihop reprogramming* using the wireless medium which reprograms the nodes as they are embedded in their sensing environment.

It is essential that the code update be 100 percent reliable and reach all the nodes that it is destined for. Since the performance of the sensor network is greatly degraded, if not reduced to zero, during reprogramming, it is essential to minimize the time required to reprogram the network. Also, as the sensor nodes have limited battery power, energy consumption during reprogramming should be minimized. Since reprogramming time and energy depend chiefly on the number of radio transmissions, a reprogramming system should minimize the amount of information that needs to be wirelessly transmitted during reprogramming. Reprogramming is done recurrently and transfers much larger data than that transmitted during regular communication of the sensed data. Hence, resource consumption of reprogramming is an important concern.

In practice, software running on a node evolves, with incremental changes to functionality, or modification of the parameters that control current functionality. So the difference between the currently executing code and the new code is often much smaller than the entire code. This makes incremental reprogramming attractive because only the changes to the code need to be transmitted and the entire code can be reassembled at the node from the existing code and the received changes. The goal of incremental reprogramming is to transfer a small *delta* (difference between the old and the new software) so that reprogramming time and energy can be minimized.

The design of incremental reprogramming on sensor nodes poses several practical challenges. A class of operating systems, that includes the widely used TinyOS [1], does not support dynamic linking of software components on a node. This rules out a straightforward way of transferring just the components that have changed and linking them in at the node. The second class of operating systems, represented by SOS [2] and Contiki [3], do support dynamic linking. However, they do not allow changes to the kernel modules. Also, the specifics of the position independent code strategy employed in SOS-1.x limits the kinds of changes to a module that can be handled. In Contiki and SOS-2.x, the requirement to transfer the symbol and relocation tables to the node to support runtime linking increases the amount of traffic that needs to be disseminated through the network.

In [4], we presented an incremental reprogramming system called Zephyr that supports fully functional incremental reprogramming in sensor networks. It does not

- *R.K. Panta is with the AT&T Research Labs, 180 Park Avenue, Building 103, Room E231, Florham Park, NJ 07932. E-mail: rpanta@research.att.com.*
- *S. Bagchi is with the Department of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, Room 329, Electrical Engineering Building, West Lafayette, IN 47907. E-mail: sbagchi@purdue.edu.*

Manuscript received 27 Jan. 2011; revised 21 Sept. 2011; accepted 28 Jan. 2012; published online 2 Feb. 2012.
Recommended for acceptance by Y. Liu.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2011-01-0049.
Digital Object Identifier no. 10.1109/TPDS.2012.55.

require dynamic linking on the node and does not transfer symbol and relocation tables. Zephyr generates the delta by comparing the two executables (called *byte-level comparison*) using an optimized version of the Rsync algorithm [5]. In order to increase the similarity between the two versions of the software to produce small delta, Zephyr uses one level of indirection for function calls to mitigate the effects of function shifts. Each function call is redirected to a fixed location in the program memory where the actual call to the function is made.

In this paper, we identify two issues with Zephyr in particular and incremental reprogramming in general—1) Function call indirections decrease the program execution speed. Although one such indirection increases the latency of a single function call by only few clock cycles (e.g., 8 clock cycles on the AVR platform [6]), the latency for each function call is experienced continually by a sensor node because typical sensor network applications execute in a loop, invoking a given function call repeatedly. Increase in latency in performing processing tasks means less amount of time for the sensor nodes to sleep causing the energy consumption to increase (although this energy consumption is small compared to that of radio communications). Further, some sensor networks are deployed in applications needing control and actuation. These typically have tight bounds on the time by which the control or the actuation commands need to be sent [7]. Therefore, the time saving due to removing the indirection may be important in such scenarios. Furthermore, by avoiding the indirection table, Hermes reduces the program memory usage, which can be an issue for some applications that need to run on devices with little program memory. 2) Function call indirections do not handle the increase in delta size due to movement of the global data variables. As the user software is changed, positions of the global variables may change and the instructions which refer to those variables may change as well between the two versions of the software. This causes a huge increase in the size of the delta.

For a wide range of software change cases that we experimented with, we found that the global variable shifts increase the delta size by 1369.56 percent on average. The increase in the size of the delta due to the relocation of the global variables depends on the number of global variables that are shifted in memory due to software modification and the number of instructions that refer to the shifted variables. From our experiments, we find that the practical software changes generally cause many global variables to be shifted. These problems exist in *all* protocols that use function call indirections and in *all* existing reprogramming protocols.

In this paper, we present a fully functional incremental reprogramming system called *Hermes* (messenger of gods, in Greek mythology) which solves the problems mentioned above. It uses indirection table to mitigate the effects of function shifts and performs local optimizations at the node to avoid the latency caused by such indirection. Thus, function call indirections are used to reduce the size of the delta that is transferred wirelessly, while efficient code, without indirections, is executed, after some local transformations. Hermes also reduces the size of delta (and hence reprogramming time and energy) significantly by using techniques to eliminate the effects of global variable shifts. This paper is an extended version of our previous conference paper [8] with following major differences. First, we introduce a problem associated with any incremental reprogramming approach, which, to the best of our knowledge, has not been considered by any previous work. In practical wireless sensor network deployments, a sensor node (or a group of nodes) may miss one or more incremental code updates. In such cases it will not be able to receive any future code update because the most recent previous version of the code is required by the sensor node to build the latest version of the code. This is a correctness issue, and thus even more serious than a performance degradation. We present a solution to this problem in this paper. Second, we analyze of the performance of Hermes, specifically the benefit of keeping global variables at the same addresses, between the old and the new versions of the software. One of the principal design techniques of Hermes is to eliminate the effects of relocation of global variables in memory due to software modification. We quantify this effect through a newly introduced metric. We provide values for this metric for a range of practical software update cases to show that Hermes provides a significant advantage. Third, we provide a more comprehensive comparison of Hermes with previous approaches.

We implement Hermes in TinyOS [1] and demonstrate it on real multihop testbeds as well as using simulations. Our experiments show that Deluge [9], Stream [10], protocol by Jeong and Culler [11], and Zephyr [4] need to transfer up to 201.41, 134.27, 64.75, and 62.09 times more bytes than Hermes, respectively.

Our contributions in this paper are as follows:

- Hermes avoids the latency in the user program due to the use of indirection table. The technique used for this demonstrates a new design approach for reprogramming sensor networks—optimize delta for the wireless transfer as radio transmissions are the most expensive operations in the sensor network and let the sensor nodes perform some local inexpensive optimizations to achieve execution efficiency.
- Hermes eliminates the effect of global variable shifts on the size of the delta script.
- We provide quantitative comparison among the existing protocols to show improvement of two orders of magnitude.

The rest of the paper is organized as follows: Section 2 surveys the related work. Section 3 gives a brief overview of various stages of Hermes. Section 3.2 discusses the byte-level comparison and explains why such comparison alone is not sufficient. Section 3.3 presents the application-level modifications, including the techniques to eliminate the effects of shifts in the memory locations of global variables. Section 3.4 discusses the delta distribution method. Section 3.5 explains image rebuild and load stage and explains how the latency due to function call indirections can be avoided. Section 3.6 explains how transient link or node failures can be handled. Section 3.7 explains some optimizations. Section 4 explains the testbed and the simulation setups and results. Section 5 analyzes the performance of Hermes in terms of reduction in the size of the delta script. Section 6 concludes the paper.

2 Related Work

The question of reconfigurability of sensor networks has been an important theme in the community. We discern three streams of work in this regard. First, is the class of work that provides virtual-machine abstractions on sensor nodes. Second, is the design for reconfigurability in sensor operating systems that do not support dynamic linking and loading. Third, is reconfigurability in systems that do support dynamic linking and loading. We discuss these three streams in order here.

Systems such as Mate [12], VM* [13], Darjeeling [14], and ASVM [15] provide virtual machines that run on resource-constrained sensor nodes. They enable efficient code updates, since the virtual machine code is more compact than the native code. However, they trade off, to different degrees, less flexibility in the kinds of tasks that can be accomplished through virtual-machine programs and less efficient execution than native code. Hermes can be employed to compute incremental changes in the virtual machine byte codes and is thus complementary to this class.

TinyOS [1] is the primary example of an operating system that does not support loadable program modules. There are several protocols that provide reprogramming with full binaries, such as Deluge [9], Stream [10], Freshet [16], MOAP [17], and MNP [18]. For incremental reprogramming, several researchers have computed deltas between existing and new images and send just the delta over the air to the nodes. Jeong and Culler [11] use Rsync to compute the difference between the old and new program images. However, it can only reprogram a single hop network. Furthermore, it does not mitigate the effects of function and global-variable shifts causing the delta to be large. [19] focuses on encoding and decoding of the delta and does not consider the function and global variable shifts. In [20], the Reijers and Langendoen modify Unix's diff program to create an edit script to generate the delta. They identify that a small change in code can cause a lot of address changes resulting in a large size of the delta. Since they also do not consider the function and global variable shifts, the size of the delta is very large.

Koshy and Pandey [21] have the design goal of keeping address patches to a small number. They reduce the effects of function shifts by using slop regions after each function in the application so that the function can grow. However, the slop regions lead to fragmentation and inefficient usage of the Flash and the approach only handles growth of functions up to the slop region boundary. Also, this work does not consider the effect of shifts in the positions of the global variables in memory. Marron et al. [22] present a mechanism for linking components on the sensor node without support from the underlying OS. This is achieved by sending the compiled image of only the changed components along with the new symbol and relocation tables to the nodes for dynamic linking on the nodes. This has been demonstrated only in an emulator and makes extensive use of Flash. Also, the symbol and relocation tables can grow very large resulting in large updates. To the best of our understanding, no previous work handles the issue of increased delta size due to global variable shifts. Previous works on incremental reprogramming have focused on one or some stages of the process while here we present the results of the complete multihop reprogramming process that executes on a testbed.

Reconfigurability is simplified in OSes like SOS [2], and Contiki [3] that support linkable modules. In these systems, individual modules can be loaded dynamically on the nodes. Specific challenges exist in the matter of reconfiguration in these systems. SOS-1.x uses position independent code and due to architectural limitations on common embedded platforms, the relative jumps can be only within a certain offset (such as 4 KB for the AVR platform). Contiki disseminates the symbol and relocation tables, which may be quite large (typically these tables make up 45 to 55 percent of the object file [21]). SOS and Contiki do not allow kernel changes. Hermes allows updates to any part of the software. Low-level comparison between files for determining their differences is achieved by several tools, including Rsync (which we compare here), Vdelta, and Xdelta.

R2 [23] is a recent incremental reprogramming system that generates delta using relocatable code. R2 optimizes the data structures for storing the relocation information for micro embedded devices. Unlike Hermes which uses different techniques to handle function shifts and global variable shifts, R2 uses a unified relocation entry approach for all types of data and code references. Hermes is designed to handle function and global variable shifts whereas R2 can also handle other reference instructions, e.g., *call, mov, br, etc*. However, unlike Hermes which is evaluated using an extensive set of experiments, R2 is evaluated using a small number of software change cases. One of the disadvantages of using relocatable code is that the relocation entries, that need to be sent along with the delta script, can be very large if there are large number of function calls and global variable references. In R2, a relocation entry needs to allocated for each function call and global variable reference. As we explain in the next section, Hermes needs to allocate a table entry for each function, not for each function call. Thus, generally the number of table entries for Hermes is significantly less than the number of relocation entries for R2. In systems where each function is called multiple times, the size of relocation entry and thus the delta can be very large in R2. Furthermore, Hermes does not require extra memory allocations for global variables, whereas R2 requires relocation entry for each reference to a global variable.

3 Hermes Design

In this section, we first present an overview of the various components of Hermes followed by their descriptions.

3.1 Overview of Hermes

Fig. 1 is the schematic diagram showing various stages of Hermes. First, Hermes performs two *application-level modifications* on the old and new versions of the software—one to mitigate the effect of function shifts and the other to eliminate the effect of global variable shifts. Then the two executables are compared at the byte level using an optimized Rsync algorithm [4]. This produces the delta script which describes the difference between the old and new versions of the software. Next the delta script is transmitted wirelessly to all the nodes in the network using

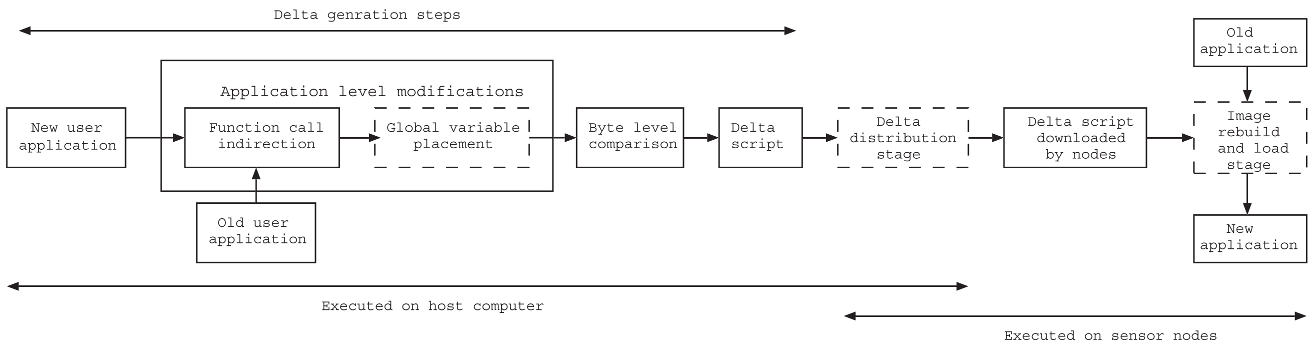


Fig. 1. Overview of Hermes: the stages with dashed rectangles are the ones which are introduced or modified by Hermes.

the delta distribution stage. Once the nodes download the delta script, they rebuild the new software using the old software and the received delta script. The sensor nodes run the newly rebuilt software by using bootloader to load it in the program memory. The stages shown in Fig. 1 are described in the following individual sections. Before explaining the application-level modifications, we first describe byte-level comparison and show why it is not sufficient and why we need application-level modifications.

3.2 Byte-Level Comparison

Hermes uses Zephyr's approach for byte-level comparison to generate the delta script. For the sake of completeness, here we provide a very brief description of this stage. Hermes computes the delta script between the two versions of the executables using modified Rsync algorithm. The delta script basically consists of COPY and INSERT commands. COPY commands tell which parts of the old software need to be copied to the new software (and where) and INSERT commands contain the bytes that are not present in the old software but need to be inserted in the new software. A complete description of Rsync algorithm and our modifications to it are explained in [4]. But the delta script produced by byte-level comparison is much larger than the actual amount of change made in the software. To see this, let us consider two software change cases.

- Case I: Changing Blink application from blinking a green LED every second to blinking it every 2 seconds. Blink is an application in TinyOS distribution that blinks an LED at the specified rate. The delta script produced with byte-level comparison is 23 bytes which is small and congruent with the amount of change made in the software.
- Case II: We added 4 lines of code to Blink. The delta script between Blink and the one with these few lines added is 2,183 bytes. The actual amount of change made in the software for this case is slightly more than that in Case I, but the delta script produced is disproportionately larger.

When a single parameter is changed in the application as in Case I, no part of the already matching binary code is shifted. All the functions start at the same location as in the old image. But with the few lines added to the code as in Case II, the functions following those lines are shifted. So all the calls to those functions refer to new locations resulting in the large delta script. Thus, we need application-level modifications to make the size of the delta script proportional to the actual amount of change made in the software.

3.3 Application-Level Modifications

Hermes uses Zephyr's approach of function call indirections to mitigate the effects of function shifts. Hermes changes the linking stage during the program compilation to redirect the function calls to the indirection table (placed at the fixed location in program memory). For example, let the application shown in Fig. 2a be changed to the one shown in Fig. 2b where functions fun1, fun2, funn are shifted from their original positions b, c, and a to b', c', and a', respectively. Hermes modifies the linking stage of the executable generating process to produce the code shown in Fig. 2c (for old image) and Fig. 2d (for new image). Here calls to functions fun1, fun2, ..., funn are replaced by jumps to fixed locations loc1, loc2, ..., locn, respectively. The segment of the program memory starting at the fixed location loc1 acts as an indirection table where the actual calls to the functions are made. When the call to the actual function returns, the indirection table directs the flow of control back to the line following the call to loc-x $(\mathrm{x} = 1, 2, \ldots, \mathrm{n})$. In Hermes, the functions that exist in both the new and old versions of the software are assigned the same slots in the indirection table. As a result, if the user program has n calls to a particular function, they refer to the same location in the indirection table and only one call in the indirection table differs between the two versions. On the other hand, if no indirection table were used, all the n calls would refer to different locations in old and new applications. Due to the use of indirection table, the delta script produced by Hermes is only 280 bytes for Case II compared to 2,183 bytes when only byte-level comparison is used. Function call indirections have been used in some wireline and wireless systems but not to reduce the delta or reprogram the sensor networks.

3.3.1 *High-Level Idea*

The basic idea behind application-level modifications is to mitigate the *structural changes* in the user program caused by the modification of the software so that the similarity between the old and new software is preserved and a small delta script is produced. Apart from function shifts, the other structural change caused by software modification is the global variable shifts. These result in all the instructions that refer to those variables to change between the two versions of the software. Note that local variables can also

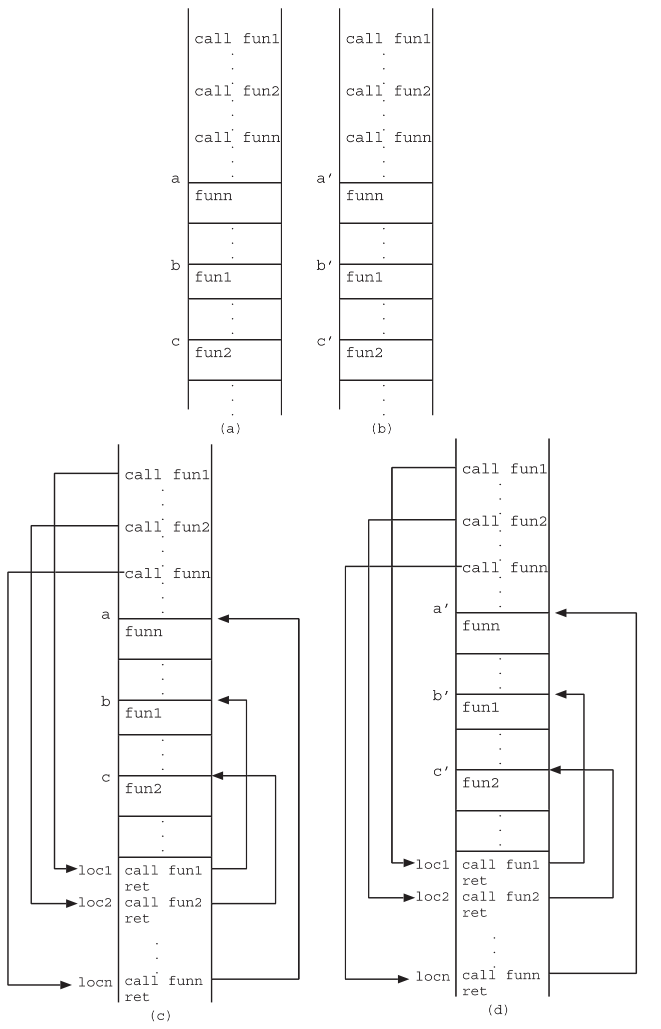


Fig. 2. (a) Old and (b) new images without indirection table (current state). Note that positions of the functions (fun1, etc.) have changed leading to changes in the call instructions. (c) Old and (d) new images with indirection table in Hermes. Note that due to the indirection table, the call instructions do not change.

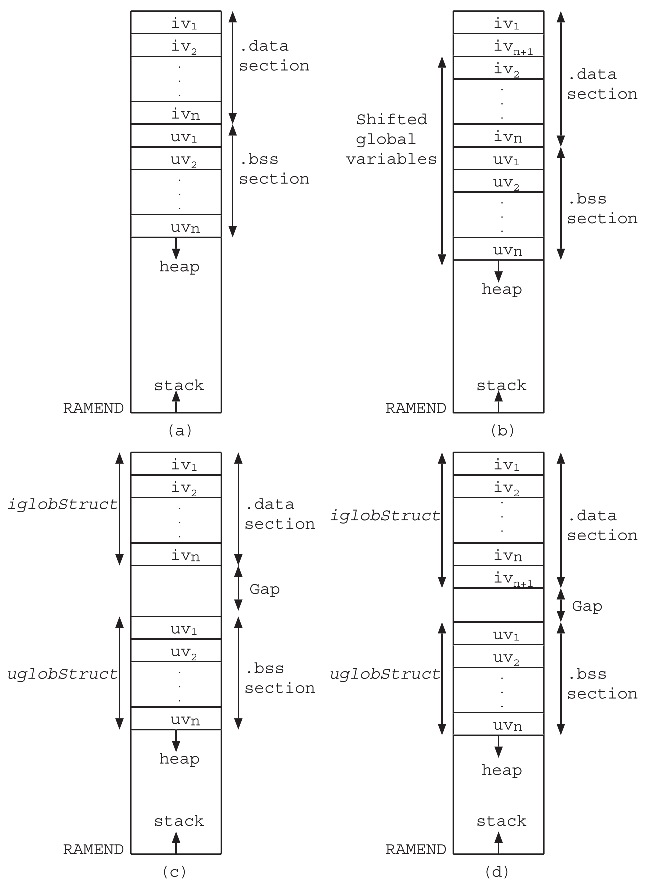


Fig. 3. Baseline RAM structures for (a) old and (b) new applications. RAM structures for corresponding (c) old and (d) new applications using Hermes.

get shifted due to change in the software, but this does not cause the instructions that refer to these variables to change. To understand this, let us see how different variables are stored in RAM. As shown in Fig. 3a, initialized global variables are stored as .data variables in RAM followed by uninitialized global variables which are stored as .bss variables. The local variables are stored in stack which grows upward from the end of RAM. Since the local variables are referred to using the addresses relative to the stack pointer, their exact locations in RAM do not affect the size of the delta script.

To see the severeness of the global variable shifts, consider an example where a global variable is added to the Blink application. In this case, the size of the delta script produced by using only indirection table is 6,090 bytes. This is disproportionately larger than the actual amount of change made in the software. The size of the delta script depends on the number of global variables that are shifted and the number of instructions that refer to those shifted variables. So, a mechanism to mitigate the effects of global variable shifts should be a very important component of application-level modifications to make the delta script size proportional to the actual amount of change made in the software.

It should be noted that the actual order of the global variables in RAM is determined by the compiler implementation, not by the order in which they are declared in the user program. So, the programmer has no control over the placement of the global variables in RAM. Since the location of global variables in RAM is dependent on the compiler specifics, one solution is to change the compiler itself and place the global variables such that the similarity in positions of the variables between the old and the new versions is maximized. But this calls for a complex modification to the core of a compiler, which in turn makes the solution difficult to port.

3.3.2 Placement of Global Variables

Since we desire a compiler-independent solution, Hermes uses the fact that members of a structure are placed in the same order in RAM as they are declared within the structure. Hermes adds one more stage (*Structure generator*) to the executable building process. If this is the first time software is being installed on the sensor nodes (i.e., no old software exists), this stage scans through the application source files and transforms the initialized global variables into members of one structure, called *iglobStruct*, and uninitialized global variables into members of another structure, called *uglobStruct*. This stage also replaces instructions that refer to the global variables by the instructions that refer to them as the corresponding members of these structures. When the software is modified, the structure generator scans through the new software to find the global variables. When such variable is found, it checks if that variable is present in the old software. If yes, it places that variable as a member of the corresponding structure (*iglobStruct* or *uglobStruct*) at the same slot in that structure as in the old software. Otherwise, it makes a decision to assign a slot in the corresponding structure for that variable (call it a *rootless* variable), but does not yet create the slot. After assigning the slots for the existing global variables, it checks if there are any empty slots in the new structure. These would correspond to variables which were present in the old software, but not in the new software. If there are empty slots and the size of the rootless variable is less than or equal to the size of the slot, Hermes assigns those slots to the rootless variables. If there are still some rootless variables without a slot, then the corresponding structure is expanded to accommodate the rootless variables. Thus, both these structures are naturally garbage collected and the structures expand on an as-needed basis.

The above method is illustrated in Fig. 3. Let default RAM structures for old and new applications be as shown in Fig. 3a and Fig. 3b, respectively. The old application has initialized global variables $iv_1, iv_2, \ldots, iv_n$ in the .data section and uninitialized global variables $uv_1, uv_2, \ldots, uv_n$ in the .bss section. Let a single initialized global variable iv_{n+1} be added to .data section due to the modification in the software and the compiler places it after iv_1 (Fig. 3b). As a result, global variables $iv_2, iv_3, \ldots, iv_n, uv_1, uv_2, \ldots, uv_n$ are shifted to new positions in RAM causing all the instructions in program memory that refer to these shifted variables to vary between the two versions of the application. This results in a large delta script. Hermes uses the two structures, *iglobStruct* and *uglobStruct*, to put .data and .bss variables, respectively as shown in Fig. 3c for the old application. Hermes also leaves some space between .data and .bss sections to allow the former to grow with less chance of the latter being straddled which would cause an undesirable shift in the uninitialized global variables. In Section 3.7, we discuss how Hermes avoids this gap. In the new application (Fig. 3d), Hermes places the added variable iv_{n+1} at the end of the .data section so that the variables which are common between the two versions of the application are located at the same locations in RAM. So the instructions referring to the global variables that exist in both the versions do not change resulting in a small delta script.

The problem of filling free slots with rootless variables of various sizes is similar to knapsack problem, which is known to be NP-Complete [24]. Hermes uses a greedy approach to tackle this problem. It is possible to achieve better performance in terms of reduction in the number or size of free slots by replacing the greedy approach with clever alternative schemes. Our current implementation uses the greedy approach because it is simple and easy to implement and the performance is good enough for our use cases. If an alternative approach is used, the tradeoff between performance gain and complexity of implementation should be considered.

These changes in Hermes are transparent to the user. She does not need to change the way she programs. Hermes applies these changes during the executable generation process when the user invokes program compilation.

With this approach, the size of the delta script produced by Hermes for the case where one global variable was added to Blink application is 156 bytes compared to 6,090 bytes when only indirection table is used (as in Zephyr). In other words, with the addition of the structure generator to the application-level modification stage, the size of the delta script is significantly reduced making it proportional to the actual amount of change made in the software.

It should be noted that function and global variable shifts require different solutions. The former is handled by calling functions through a level of indirection; the latter is handled by preserving the similarity in the order in which the variables are placed in RAM. Static placement of functions would waste some memory for interfunction space. Global variable access through indirection would result in code expansion and runtime overhead.

3.4 Delta Distribution Stage

For wirelessly distributing the delta script, Hermes uses the approach from Stream [10] with some modifications. The core data dissemination method of Stream is the same as in Deluge. Deluge uses a monotonically increasing version number, segments the binary code image into pages, and pipelines the different pages across the network. The code distribution occurs through a three-way handshake of advertisement [25], request, and code broadcast between neighboring nodes which ensures reliability in the face of wireless link failures. Unlike Deluge, Stream avoids transferring the entire reprogramming component every time code update is done. The reason behind this requirement in Deluge is that the reprogramming component needs to be running on the sensor nodes all the time so that the nodes can be receptive to future code updates and these nodes are not capable of multitasking (running more than one application at a time). Stream solves this problem by storing the reprogramming component in the external flash and running it on demand—whenever reprogramming is to be done.

Distinct from Stream, Hermes divides the external flash as shown in the right side of Fig. 4. The reprogramming component and delta script are stored as image 0 and image 1, respectively. Images 2 and 3 are the user applications—one old version and the other current version which is created from the old image and the delta script as discussed in Section 3.5. The protocol works as follows:

1. Let image 2 be the current version (v_1) of the user application. Initially, all nodes in the network are running image 2. At the host computer, delta script is generated between the old image (v_1) and the new image (v_2).
2. The user gives the command to the *base node* (node physically attached to the host computer) to reboot all nodes in the network from image 0 (reprogramming component).
3. The base node broadcasts the reboot command and itself reboots from the reprogramming component.
4. The nodes receiving the reboot command from the base node rebroadcast the reboot command and

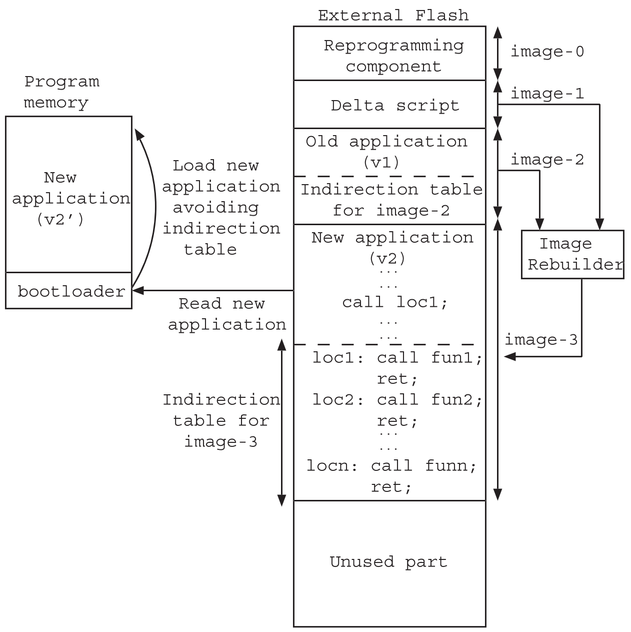


Fig. 4. Image rebuild and load stage. The right side shows the structure of external flash in Hermes.

themselves reboot from the reprogramming component. This is controlled flooding because each node broadcasts the reboot command only once. Finally, all nodes in the network are executing the reprogramming component.

5. The user then injects the delta script to the base node. It is wirelessly transmitted to all nodes in the network using the usual three-way handshake of advertisement, request, and code broadcast as in Deluge. Note that unlike Stream and Deluge which transfer the application image itself, Hermes transfers the delta script only.
6. All nodes receive the delta script and store it as image 1. Reprogramming a heterogeneous network can be supported relatively easily on top of Hermes by storing multiple application image pairs (old and new) one for each class of nodes. The instruction to reboot from a specific image is sent separately to each class of nodes.

3.5 Image Rebuild and Load Stage

After the nodes download the delta script, they rebuild the new image using the script (stored as image 1 in the external flash) and the old image (stored as image 2 in the external flash). The image rebuilder stage consists of a *delta interpreter* which interprets the COPY command by copying the specified number of bytes from the specified location in the old image to the specified location in the new image. All these locations are specified in the COPY command of the delta script. The interpreter inserts the bytes present in the INSERT command at the specified location in the new image. The new image is stored as image 3. The bootloader then loads the new software from image 3 of the external flash to the program memory (Fig. 4). In the next round of reprogramming, image 3 becomes the old image and the newly rebuilt image is stored as image 2.

Next, we describe the processing at the bootloader when creating the executable image.

Avoiding latency due to indirection table. As mentioned earlier, Hermes uses Zephyr's approach of function call indirections to mitigate the effects of the function shifts. Use of one extra level of indirection increases the latency of the user program.

To solve this problem, we observe that there are two conflicting requirements: we need indirection table to reduce the size of the delta script and we need to remove any indirection for optimized execution speed. We solve this by having the sensor nodes store the application with indirection table in the external flash, but we change the bootloader to avoid using indirection table. As shown in Fig. 4, when the bootloader loads the new image (image-3) from external flash to program memory, it eliminates the indirection by using the exact function address from the indirection table. For example, in Fig. 4, when the bootloader reads *call loc1*, it finds from the indirection table that the actual target address for this call instruction is *fun1*. So when writing to program memory, it writes *call fun1* instead of *call loc1*. Thus as shown in Fig. 4, the application image in program memory (v_2') is different from that in the external flash (v_2) in that it does not use indirection table. In this way, the sensor nodes still possess the program image with the indirection table in the external flash which helps to rebuild the new image in future, and yet the currently running instance of the program image does not use the indirection table and is thus optimized for execution speed. With this, we put forward a new idea for reprogramming sensor nodes—since radio transmissions are the most expensive operations, optimize for the transfer and let the sensor nodes perform some inexpensive local operations to optimize for execution speed.

3.6 Failure Handling

In this section, we discuss one of the problems associated with all incremental reprogramming approaches, which, to the best of our knowledge, has not been considered by any previous work. Let us consider a situation where a node n_1 (or a set of nodes) goes into a *disconnection state* for some time. A disconnection state means all the links of a node have failed, in a transient manner. In practical sensor network deployments, nodes may get disconnected from the rest of the network for some time due to various reasons, such as time-varying nature of wireless channels, changing environmental conditions, transient software or hardware failures, battery outages, etc.

Let $\Delta_{i,i+1}$ represent the delta script that can be used to build the application code of version v_{i+1} from the code of version v_i, i.e., $v_i + \Delta_{i,i+1} = v_{i+1}$. Before going to the disconnection state, let us suppose that the node n_1 had v_j version of the application code. When n_1 comes out of disconnection state, let us suppose that it missed $\Delta_{j,j+1}, \dots, \Delta_{k-1,k}$ versions of the delta script, where $k - j \geq 1$. In other words, n_1 missed one or more incremental code updates while it was in the disconnection state. After coming out of disconnection state, n_1 needs v_{k-1} version of the application code and $\Delta_{k-1,k}$ version of the delta script to build the code of version v_k ($v_{k-1} + \Delta_{k-1,k} = v_k$). It can download $\Delta_{k-1,k}$ from its neighbors. However, n_1 does not possess v_{k-1} version of the user application. Thus it cannot build the latest version of the application. Note that this is a correctness problem, not only a performance issue.

Hermes provides the following intuitive solution to this problem. Note that the actual dissemination of the delta script occurs using three-way handshake of *advertisement request data* as in Deluge [9]. Nodes periodically advertise their metadata consisting of the version number of the images that they currently possess. When a node receives an advertisement message with $\Delta_{i-1,i}$ version of the delta script, it checks if it has $\Delta_{i-2,i-1}$ version of the delta script. If yes, then as usual it requests the $\Delta_{i-1,i}$ version of the delta script (i.e., image-1), downloads it, and rebuilds the new image using the downloaded delta script and the old image (version v_{i-1}). Otherwise, if it has $\Delta_{j-1,j}$ version of the delta script and $i-j>1$, then instead of requesting the delta script, it requests the entire image of the new application. Note that this approach works well even if a cluster of nodes in a geographical vicinity misses one or more recent delta script downloads. Some of those *out-of-date* nodes may not have any *up-to-date* neighbor. But as long as at least one out-of-date node has a functioning link with at least one up-to-date node, all the out-of-date nodes will eventually get the latest version of the entire image.

Obviously downloading the entire new image is more costly than downloading just the delta script. But Hermes compromises performance for correctness. However, it should be noted that the excessive radio transmissions for downloading the entire program image are localized only in the neighborhood of the node(s) which has missed n recent code updates. An alternative solution would be to send the last n deltas. However, this poses several problems and is generally more energy expensive than downloading the new image because of the following reason. In this alternative scheme, each node needs to decide the number of previous deltas that it needs to store in case some node needs the set of deltas from itself. Obviously, it cannot store all past deltas and since the node disconnection period can be arbitrarily large, the finite number of deltas stored in the nodes may not be sufficient. In such cases, the last n deltas will have to be delivered from the base station to the required sensor node through a series of intermediate nodes. This can be more expensive in terms of energy consumption than just downloading the entire new image from an immediate neighbor.

3.7 Avoiding Empty Space between .data and .bss Sections

One drawback of the scheme outlined above is that we need to leave some empty space between .data and .bss variables in RAM to allow for .data variables to grow in future. If this space is too small, the probability of .data variables extending beyond the empty space when the software is modified becomes high, causing the .bss variables to shift. As a result, the delta script becomes large. To avoid this situation, we need to leave sufficiently large space between .data and .bss variables in RAM. But RAM is a limited resource on the sensor nodes. For example, mica2, and micaz motes have 4 KB RAM. Next, we explain how we solve this problem in Hermes.

One possible solution is to leave a large space between .data and .bss sections while compiling the application on the host computer, generate the delta script on the host computer, distribute the delta script to all the sensor nodes in the network and change the bootloader running on the sensor nodes to avoid that space. When the bootloader loads the application from external flash to the program memory, it can change the instructions that refer to .bss variables by subtracting *gapSize* from the addresses used by these instructions where *gapSize* is the size of the empty space between .data and .bss variables. Because of the complex addressing schemes on the common sensor node platforms, an algorithm with some control flow analysis is needed. Given the tight computational and memory constraints of the sensor nodes, this may not be feasible.

To solve this problem, Hermes uses two different approaches for Von-Neumann (e.g., msp430 platform [26]) and Harvard (e.g., AVR platform [6]) architectures, respectively. In Von-Neumann architecture, a single bus is used as the instruction and the data bus. Program memory (where program code is stored) and RAM (where global variables, stack, and heap are stored) share the same logical address space and therefore the same mode for addressing the two kinds of memory. As a result, we can move .bss variables from RAM to program memory and avoid the space between the .data and .bss variables in RAM. We implemented this approach on TMote [27] (msp430 platform) sensor nodes. Note that program memory is larger than RAM on the sensor nodes (e.g., TMote has 10 KB RAM and 48 KB program memory). The reprogramming system that we use occupies only about 25 KB of program memory and hence enough space is available for .bss variables in program memory. However, it should be noted that for nonread-only variables, writing to program memory involves time and energy overhead. Furthermore, writing to flash memory may require bloc erase and thus extra energy overhead. Also, there is a practical limit on the number of flash erase/write cycles for such memory. The scheme that we describe next for the Harvard architecture does not suffer from these problems and can also be used for the Von-Neumann architecture. This scheme leaves a small bounded space (10 bytes in our current implementation) between .data and .bss sections.

In Harvard architecture, program memory and RAM lie in separate address spaces. So, if we move .bss variables to program memory, we need to change all the instructions that use data bus to refer to .bss variables with different addressing modes to use the instruction bus instead. This increases the complexity of the implementation. Furthermore, even if .bss variables are stored in program memory, we can write to those locations only from restricted areas of the program memory (e.g., bootloader section) due to memory protection. This would disallow references to the .bss variables from general-purpose user programs. Thus for Harvard architecture, when the application is compiled on the host computer, Hermes leaves a small space between the two sections in RAM. If .data section expands beyond this space, we move *only* those .bss variables which are straddled by the .data section expansion to the end of the .bss section. For our mica2 [28] experiments, we leave an empty space of 10 bytes between .data and .bss sections. This is not a significant number because mica2 (and also micaz) nodes have 4 KB RAM.

4 Experiments and Results

To evaluate the performance of Hermes, we considered following software change scenarios for TinyOS applications.

- Case 1: Blink to Blink with a global variable added.
- Case 2: Blink to CntToLeds.
- Case 3: Blink to CntToLedsAndRfm.
- Case 4: CntToLeds to CntToLedsAndRfm.

CntToLeds is an application that displays the lowest 3 bits of the counting sequence on the LEDs. In addition, CntToLedsAndRfm transmits the counting sequence over the radio. To evaluate the performance of Hermes with respect to natural evolution of the real world software, we considered a real world sensor network application called eStadium [29] deployed in Ross Ade football stadium at Purdue. eStadium applications provide safety and security functionality, infotainment features such as coordinated cheering contests among different parts of the stadium using the microphone data, information to fans about lines in front of concession stands, etc. We considered a subset of the changes that the software had actually gone through, during various stages of refinement of the application.

- Case A: An application that samples battery voltage and temperature from MTS310 [28] sensor board to one where few functions are added to sample the photo sensor also.
- Case B: We decided to use opaque boxes for the sensor nodes. So, few functions were deleted to remove the light sampling features.
- Case C: In addition to temperature and battery, we added the features for sampling all the sensors on the MTS310 board except light (e.g., microphone, accelerometer, magnetometer).
- Case D: Same as case C but with the addition of a feature to reduce the frequency of sampling battery voltage.
- Case E: Same as case D but with the addition of a feature to filter out microphone samples (considering them as noise) if they are greater than some threshold value.

Case 1, Case D, and Case E are small changes; Case 2 is a moderate change; Case A, Case B, and Case 4 are large changes; Case 3 and Case C are huge changes in the software.

4.1 Size of Delta Script

Fig. 5 shows the number of bytes required to be transmitted for reprogramming by Deluge, Stream, Rsync, Zephyr, and Hermes for the software change cases mentioned above. For Deluge and Stream, the size of the information to be transmitted is the size of the binary image while for the other schemes it is the size of delta script. A small delta script translates to smaller reprogramming time and energy due to less number of packet transmissions over the network and less number of flash writes on the node. For small changes in software (like Case 1, Case D, and Case E), the incremental reprogramming protocols perform much better. Note that Rsync is the approach by Jeong and Culler [11]. We find that Hermes significantly reduces the size of the delta script compared to other approaches. Deluge,

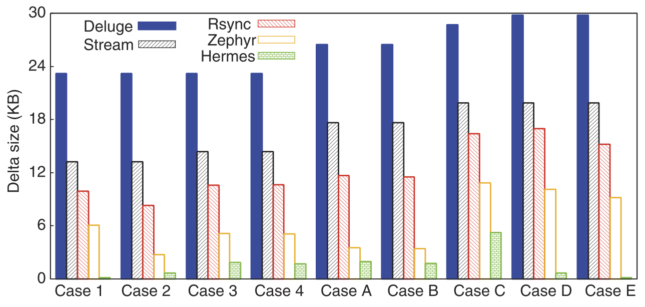


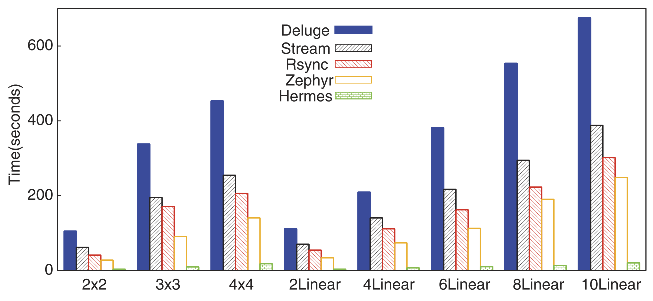
Fig. 5. Comparison of number of bytes that various approaches need to transmit for the nine software change examples.

Stream, Rsync, and Zephyr take up to 201, 134, 64, and 62 times more bytes than Hermes, respectively.

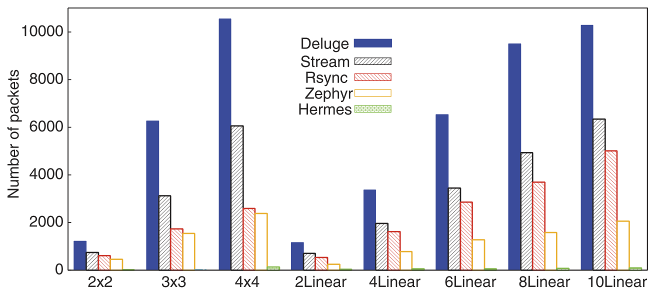
Koshy and Pandey [21] use slop region after each function to avoid the effects of the function shifts. Hence the delta script for their best case (when none of the functions expand beyond the assigned slop regions) will be same as that of Zephyr. But even in their best case scenario, the program memory is fragmented and the ratios of Hermes to [21] would be identical to that of Hermes to Zephyr. Fig. 5b shows that [21] requires to transmit 1.79 to 62.09 times more information than Hermes for reprogramming. This huge advantage shows the importance of our approach to eliminate the effects of global variable shifts. The exact amount of advantage of Hermes over Zephyr is directly proportional to the number of global variables that are shifted in Zephyr due to change in the software and the number of times those shifted variables are referred to in the program code. For example, the addition or deletion of .data variables results in more reduction in the size of the delta script by Hermes compared to Zephyr than the .bss variables. We refer to Jeong and Culler [11] as Rsync because their approach is to generate the difference using Rsync. Their approach compares the two executables without any application-level modifications. The ratios of Rsync to Hermes greater than 1 show the importance of the Rsync optimization [4] and the application-level modifications (both function call indirections and global variable placements). Rsync [11] approach needs to transfer 3.14 to 64.75 times more bytes than Hermes.

4.2 Testbed Experiments

We perform testbed experiments using Mica2 [28] nodes for grid and linear topologies. For each network topology, we define neighbors of a node n_1 as those nodes which are adjacent to that node n_1 in the specific topology. For the grid network, the transmission range R_{tx} of a node satisfies $\sqrt{2}d < R_{tx} < 2d$, where d is the separation between the two adjacent nodes in any row or column of the grid. The linear networks have the nodes with the transmission range R_{tx} such that $d < R_{tx} < 2d$ where d is the distance between the adjacent nodes. Due to fluctuations in transmission range, occasionally a nonadjacent node will receive a packet. In our experiments, if a node receives a packet from a nonadjacent node, it is dropped. This kind of software topology control has been used in other works also [30], [31]. For the grid network, a node situated at one corner of the grid acts as the base node while the node at one end of



(a)



(b)

Fig. 6. Comparison of reprogramming (a) time and (b) energy between Hermes and other approaches for various grid and linear networks for Case 1.

the line is the base node for linear networks. We provide quantitative comparison of Hermes with Deluge [9], Stream [10], Rsync (Jeong and Culler [11]), and Zephyr [4]. Note that Jeong and Culler [11] reprogram only nodes within one hop of the base node, but we used their approach on top of multihop reprogramming protocol to provide a fair comparison. We perform these experiments for grids of size 2×2 to 4×4 and linear networks of size 2 to 10 nodes. The results presented here are the average over these grid and linear networks.

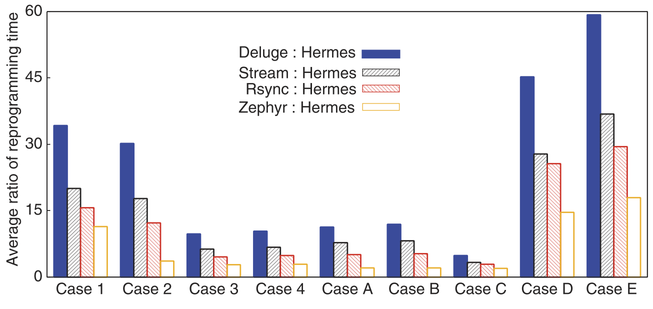
4.2.1 *Reprogramming Time and Energy*

Time to reprogram the network is the sum of the time to download the delta script and the time to rebuild the new image. We used the approach of [31] to measure the network reprogramming time. Fig. 6 compares reprogramming time and total number of bytes transmitted by various approaches for Case 1. Fig. 7a compares the ratio of reprogramming times of other approaches to Hermes for all software change cases. As expected, Hermes outperforms the nonincremental reprogramming protocols Deluge and Stream significantly. Hermes is also 2.88 to 29.51 times faster than Rsync [11]. This illustrates that application-level modifications that Hermes applies are very important in reducing the time to reprogram the networks. As mentioned above, the best case scenario for Koshy and Pandey [21] is same as that of Zephyr. Hermes is 2.01 to 17.93 times faster than Zephyr. This shows how Hermes' technique to eliminate the effects of the global variable shifts translates into speeding up the reprogramming process. To see the significance of these improvements, let us consider Case E. Deluge, Stream, Rsync, Zephyr, and Hermes took 648.68, 347.19, 299.78, 196.06, 195.06, and 14.24 seconds, respectively to reprogram the 4×4 grid. Note that Hermes is most effective for small or moderate software change cases (like Case 1, Case 2, Case D, and Case E) which are more likely to happen in practice. The time to rebuild the new image at the sensor node depends on the size of the delta script, but is small compared to the total reprogramming time. In all these experiments, the image rebuild time even on the resource-constrained sensor nodes is less than 6 seconds which is small compared to the total reprogramming time (in the order of several minutes).

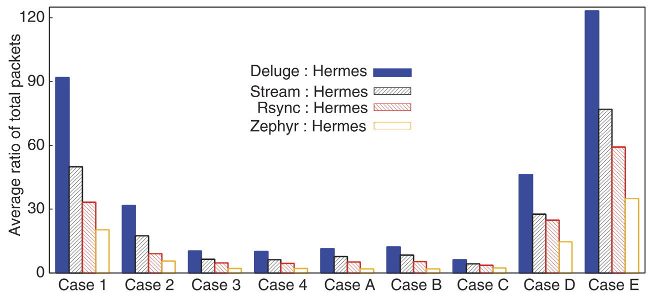
Among the various factors that contribute to the energy consumed during reprogramming, two important ones are the amount of radio transmissions and the number of flash writes (the downloaded delta script is written to the external flash). Since both of them are proportional to the number of packets transmitted in the network during reprogramming, we take the total number of packets transmitted by all nodes in the network as the measure of energy consumption. Fig. 6b compares the total number of packets transmitted by all nodes in the network using Hermes with other schemes for the above mentioned grid and linear networks. Like reprogramming time, Hermes reduces the number of packets transmitted during reprogramming significantly compared to other approaches. As indicated by the ratios of Zephyr to Hermes, the elimination of the global variable shifts results in a very large savings (2.01 to 35.12 times) in energy.

4.2.2 *Execution Speed*

In order to demonstrate latency improvement for Hermes due to the use of the technique to avoid the indirection table, we considered a typical sensor network application which operates in a loop with each run of the loop consisting of *work* and *sleep* periods. In the work period, a node samples all the sensors on MTS310 sensor board [28],

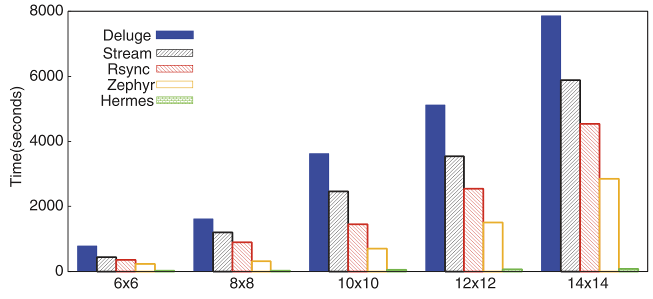


(a)

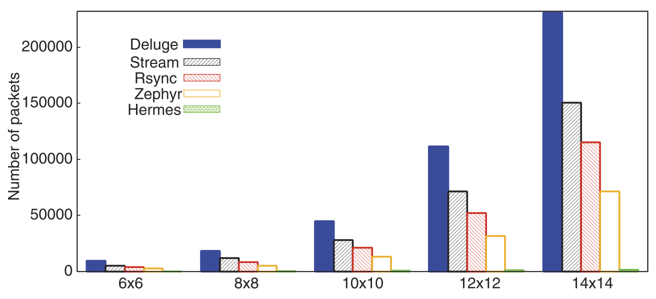


(b)

Fig. 7. Average ratio of (a) reprogramming time and (b) total packets transmitted by other approaches to Hermes.



(a)



(b)

Fig. 8. Simulation results: Comparison of reprogramming (a) time and (b) energy between Hermes and other approaches for various grid networks for Case E.

processes the sampled data (calibrate the sensor readings, filter out noisy microphone readings, etc.) and sends the data to the cluster head. In the sleep period, the node goes to sleep to save energy. All function calls happen in the work period. We find that Hermes reduces the work period by about 3.7 ms per run of the loop compared to Zephyr. More importantly, the savings in latency increases linearly with the number of loop iterations. However, it should be noted that the exact savings depend on the number of function calls executed by a given application during the work period.

4.3 Simulation Results

We perform TOSSIM [32] simulations on grid networks of varying size (up to 14×14) to demonstrate the scalability of Hermes and to compare it with other schemes. Fig. 8 shows the reprogramming time and number of packets transmitted during reprogramming for Case E. We find that Hermes is up to 94, 70, 54, 34 times faster than Deluge, Stream, Rsync, and Zephyr, respectively. Also, Deluge, Stream, Rsync, and Zephyr transmit up to 149, 97, 74, and 46 times more number of packets than Hermes, respectively. Hermes is as scalable as Deluge since none of the changes in Hermes affects the three-way code dissemination handshake or changes with the scale of the network. All application-level modifications are performed on the host computer and the image rebuilding on each node does not depend upon the number of nodes in the network. These simulation results also show that as the network grows larger, Hermes' advantage over existing protocols increases. This happens because with the increase in the network size, the existing protocols face more contention and collisions as they need to transfer more bytes than Hermes.

5 Analysis

A principle design technique of Hermes is to make sure that if a global variable is present in both the old and the new versions of the software, it is placed at the same location in memory. As a result, the instructions in the program code that refer to these common variables do not change between the two versions of the software. In existing incremental reprogramming approaches, these common variables may not be placed in the same memory location, causing the delta script to be large. Thus, the effectiveness of Hermes depends on the number of global variables that are shifted in the new version of the software (without the use of Hermes) and the number of times those shifted variables are referenced in the program code.

In this section, we analyze the performance of Hermes in terms of reduction in the size of the delta script by keeping the common global variables in the same memory location in the new version of the software as in the old one. First, we define a few terms. Let two instructions—one in the old and the other in the new version of the software—referencing the same global variable be called *identical references*. Note that in the baseline case, which may not place a common global variable at the same memory location, the actual variable addresses in the identical instructions may be different. But in Hermes, the variable addresses in the identical references are made identical. We use an attribute called *Preserved Similarity Index (PSI)* to quantify the amount of similarity preserved by Hermes in the new version of the software with respect to the old version. We define PSI as

$$PSI = \frac{R_S}{R}, \quad (1)$$

where R_S is the number of identical references in the new software that would have different variable addresses than those in the old software if Hermes were not used. R is the total number of identical references in the old and the new images. Note that $R_S \leq R$ and hence, the PSI value lies between 0 and 1.

A high PSI value means that the amount of similarity preserved by Hermes is also high. For example, if $\text{PSI} = 1$, then without Hermes, all the identical references in the old and new images use different global variable addresses, whereas with Hermes, they have the same address. This translates to a large advantage due to the use of Hermes. Note that $\text{PSI} = 0$ means that even without Hermes, the identical references would have same global variable address because the memory locations of the corresponding global variables are not changed by the software modification. Hence, in this case, there is no advantage due to Hermes. Fig. 9 shows the PSI values for different software change cases discussed in Section 4. Note that for many cases, $\text{PSI} > 0.9$. This suggests that many software change cases cause a lot of global variables to be shifted. Hence, without avoiding the effects of such shifts, the number of identical references with different global variable addresses in the two versions of the software is high. This causes the delta script to be large. This also explains the observation

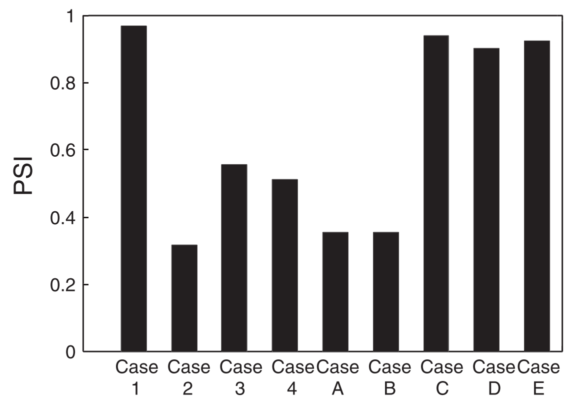


Fig. 9. Preserved similarity index for different software change cases.

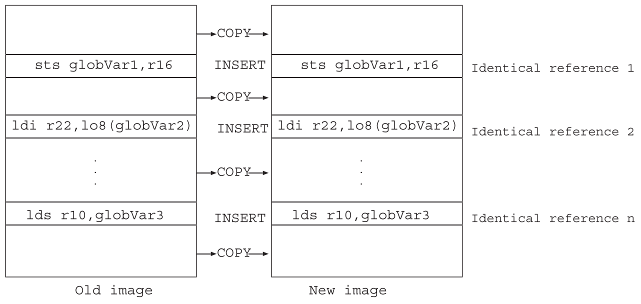


Fig. 10. Without Hermes, the difference between the *identical code segments* require $n+1$ COPY commands and n INSERT commands in the delta script. Here *ldi*, *sts*, and *lds* are different instructions that refer to the global variables.

that byte-level comparison alone is not sufficient, and we need to enhance the structural similarity between the two versions of the software by keeping locations of global variables identical to create a small delta script.

Next, we quantify the effect of global variable shift on the size of the delta script. As shown in Fig. 10, let us consider two code segments in the old and the new versions of the software, which are *identical* except that the global variable address of n identical references are different. Without Hermes, we need $n+1$ COPY commands and n INSERT commands to describe the difference between these code segments. The delta script would therefore be $(n+1)*7+n+7=14n+7$ bytes long (since each COPY and INSERT command takes 7 bytes assuming that the address of a global variable is 2 bytes). With Hermes, only one COPY command is required, which is 7 bytes long. In other words, Hermes decreases the size of the delta script by (approximately) 14 times the number of times that the shifted global variables are referenced in the new program code. Note that this analysis only quantifies the reduction in the size of the delta script due to the shifted global variables.

6 Conclusion

In this paper, we presented a multihop incremental reprogramming system called Hermes that minimizes the reprogramming overhead by reducing the size of the delta script that needs to be disseminated through the network. To the best of our knowledge, we are the first ones to use techniques to mitigate the effects of global variable shifts and avoid the latency caused by function call indirections for incremental reprogramming of sensor networks. Our scheme can be applied to systems like TinyOS which do not provide dynamic linking on the nodes. Our experimental results show that for a large variety of software change cases, Hermes significantly reduces the volume of traffic that needs to be disseminated through the network compared to the existing techniques. This leads to reductions in reprogramming time and energy. As part of our future work, we plan to use multiple code sources and multiple channels to speed up reprogramming.

References

[1] http://www.tinyos.net, 2012.

[2] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, “SOS: A Dynamic Operating System for Sensor Networks,” *Proc. Third Int'l Conf. Mobile Systems, Applications, and Services (MobiSys)*, pp. 163-176, 2005.

[3] A. Dunkels, B. Gronvall, and T. Voigt, “Contiki-A Lightweight and Flexible Operating System For Tiny Networked Sensors,” *Proc. IEEE 29th Ann. Int'l Conf. Local Computer Networks*, pp. 455-462, 2004.

[4] R. Panta, S. Bagchi, and S.P. Midkiff, “Zephyr: Efficient Incremental Reprogramming of Sensor Nodes Using Function Call Indirections and Difference Computation,” *Proc. USENIX Ann. Technical Conf.*, 2009.

[5] A. Tridgell, “Efficient Algorithms for Sorting and Synchronization,” PhD thesis, Australian Nat'l Univ., 1999.

[6] http://www.atmel.com, 2012.

[7] J. Koo, R. Panta, S. Bagchi, and L. Montestruque, “A Tale of Two Synchronizing Clocks,” *Proc. Seventh ACM Conf. Embedded Networked Sensor Systems (SenSys)*, pp. 239-252, 2009.

[8] R. Panta and S. Bagchi, “Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks,” *Proc. IEEE INFOCOM*, pp. 639-647, 2009.

[9] J. Hui and D. Culler, “The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale,” *Proc. ACM Conf. Embedded Networked Sensor Systems (SenSys)*, pp. 81-94, 2004.

[10] R. Panta, I. Khalil, and S. Bagchi, “Stream: Low Overhead Wireless Reprogramming for Sensor Networks,” *Proc. IEEE INFOCOM*, pp. 928-936, 2007.

[11] J. Jeong and D. Culler, “Incremental Network Programming for Wireless Sensors,” *Proc. IEEE First Ann. Comm. Soc. Conf. Sensor and Ad Hoc Comm. and Networks (SECON)*, pp. 25-33, 2004.

[12] P. Levis and D. Culler, “Maté: A Tiny Virtual Machine for Sensor Networks,” *ACM SIGOPS Operating Systems Rev.*, vol. 36, pp. 85-95, 2002.

[13] J. Koshy and R. Pandey, “VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks,” *Proc. Third Int'l Conf. Embedded Networked Sensor Systems (SenSys)*, pp. 243-254, 2005.

[14] N. Brouwers, K. Langendoen, and P. Corke, “Darjeeling, A Feature-Rich VM for the Resource Poor,” *Proc. ACM Seventh Conf. Embedded Networked Sensor Systems (SenSys)*, pp. 169-182, 2009.

[15] P. Levis, D. Gay, and D. Culler, “Active Sensor Networks,” *Proc. Second USENIX/ACM Symp. Networked Systems Design and Implementation (NSDI)*, 2005.

[16] M. Krasniweski, R. Panta, S. Bagchi, C. Wang, and W. Chappel, “Energy-Efficient On-Demand Reprogramming of Large-Scale Sensor Networks,” *ACM Trans. Sensor Networks (TOSN)*, vol. 4, pp. 1-38, 2008.

[17] T. Stathopoulos, J. Heidemann, and D. Estrin, “A Remote Code Update Mechanism for Wireless Sensor Networks,” Technical Report CENS-TR-30, Univ. of California, 2003.

[18] S. Kulkarni and L. Wang, “MNP: Multihop Network Reprogramming Service for Sensor Networks,” *Proc. IEEE 25th Int'l Conf. Distributed Computing Systems (ICDCS)*, pp. 7-16, 2005.

[19] P. Rickenbach and R. Wattenhofer, “Decoding Code on a Sensor Node,” *Proc. IEEE Fourth Int'l Conf. Distributed Computing in Sensor Systems (DCOSS)*, pp. 400-414, 2008.

[20] N. Reijers and K. Langendoen, “Efficient Code Distribution in Wireless Sensor Networks,” *Proc. ACM Second Int'l Conf. Wireless Sensor Networks and Applications*, pp. 60-67, 2003.

[21] J. Koshy and R. Pandey, "Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks," *Proc. Second European Workshop Wireless Sensor Networks (EWSN),* pp. 354-365, 2005.
[22] P. Marron, M. Gauger, A. Lachenmann, O. Minder, D. Saukh, and K. Rothermel, "FLEXCUP: A Flexible and Efficient Code Update Mechanism for Sensor Networks," *Proc. Third European Workshop Wireless Sensor Networks (EWSN),* pp. 212-227, 2006.
[23] W. Dong, Y. Liu, C. Chen, J. Bu, and C. Huang, "R2: Incremental Reprogramming Using Relocatable Code in Networked Embedded Systems," *Proc. IEEE INFOCOM,* pp. 376-380, 2011.
[24] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.
[25] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: A Self-Regulating Algorithm for Code Maintenance and Propagation in Wireless Sensor Networks," *Proc. First USENIX/ACM Symp. Network Systems Design and Implementation (NSDI),* 2004.
[26] http://www.ti.com/msp430, 2012.
[27] http://www.sentilla.com, 2012.
[28] http://www.xbow.com, 2012.
[29] http://estadium.purdue.edu, 2011.
[30] A. Kamra, V. Misra, J. Feldman, and D. Rubenstein, "Growth Codes: Maximizing Sensor Network Data Persistence," *Proc. SIGCOMM,* pp. 255-266, 2006.
[31] R. Panta, I. Khalil, S. Bagchi, and L. Montestruque, "Single versus MultiHop Wireless Reprogramming in Sensor Networks," *Proc. Fourth Int'l Conf. Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom),* pp. 1-7, 2008.
[32] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire Tinyos Applications," *Proc. First ACM Int'l Conf. Embedded Networked Sensor Systems (SenSys),* pp. 126-137, 2003.

Rajesh Krishna Panta received the BE degree in electronics engineering from Tribhuwan University, Nepal, the MS degree in information engineering from Niigata University, Japan, and the PhD degree in electrical and computer engineering from Purdue University in West Lafayette, Indiana. Since 2010, he is a member of technical staff at AT&T Research Labs. His research interests include wireless ad hoc and sensor networks, distributed systems, and applications of graph theory and stochastic geometry in network design and management. He is a member of the IEEE.

Saurabh Bagchi received the PhD degree from the Computer Science Department of the University of Illinois at Urbana-Champaign with Prof. Ravishankar Iyer at the Coordinated Science Laboratory, in 2001. He is an associate professor in the School of Electrical and Computer Engineering at Purdue University in West Lafayette, Indiana. He is serving as the program committee co-chair for the International Symposium on Dependable Systems and Networks (DSN) in 2011 and as an associate editor of the *IEEE Transactions on Mobile Computing*. He is an assistant director of the Interdisciplinary Computer Security Center at Purdue called CERIAS. His research interest includes dependability of distributed systems. The application domains for his work have come from web services, wireless networks, and high performance computing. He is a senior member of the IEEE Computer Society and the ACM. Further details of his group's work can be found on the web pages of the Dependable Computing Systems Lab (DCSL) at Purdue University.