Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks

Rajesh K. Panta, Saurabh Bagchi
Dependable Computing Systems Lab (DCSL)
School of Electrical and Computer Engineering, Purdue University
{rpanta,sbagchi}@purdue.edu

***Abstract*—Wireless reprogramming of sensor nodes is a requirement for long-lived networks due to changes in the functionality of the software running on the nodes. 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 multi-hop incremental reprogramming protocol 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 small delta. For a 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 protocol for TinyOS and 64 times less than an existing incremental reprogramming protocol by Jeong and Culler.**

I. 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*. The most relevant form of reprogramming is remote multi-hop reprogramming using the wireless medium which reprograms the nodes as they are embedded in their sensing environment. 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 amount of radio transmissions, reprogramming protocol 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 limits the kinds of changes to a module that can be handled. In Contiki, 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 protocol called Zephyr which does not 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 serious problems related 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 AVR platform [6]), it should be noted that the increase in latency accumulates as the application executes repeatedly in a loop. Increase in latency means less amount of time for the sensor nodes to sleep causing the energy consumption to increase and network lifetime to decrease. 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 change and the instructions which refer to those variables change as well between the two versions of the software. This causes a huge increase in the size of the delta. For example, 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% on average. This translates to proportionate increase in the time and energy required to reprogram the network. 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 protocol 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. Hermes also reduces the size of delta significantly by pinning down global variables to existing locations. We implement Hermes on TinyOS and demonstrate it on real multi-hop testbeds as well as using simulations. Our experiments show that Deluge [7], Stream [8], protocol by Jeong and Culler [9] 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:1) 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 expensive and let the sensor nodes perform some local inexpensive optimizations to achieve execution efficiency. 2) Hermes eliminates the effect of global variable shifts on the size of the delta script.

II. Related Work

The question of reconfigurability of sensor networks has been an important theme in the community. Systems such as Mate [10], VM* [11], and ASVM [12] 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 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 [7] and Stream [8]. For incremental reprogramming, Jeong and Culler [9] use Rsync to compute the difference between the old and new program images. However, it can only reprogram a single hop network and does not mitigate the effects of function and global variable shifts causing the delta to be large. [13] focuses on encoding and decoding of the delta and does not consider the function and global variable shifts. Koshy and Pandey [14] 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. The authors in [15] present a mechanism for linking components on the sensor node 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 multi-hop 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 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% of the object file [14]). Hermes, while currently implemented in TinyOS, can also be complementary to SOS and Contiki to upload incremental changes within a module. Low-level comparison between files for determining their differences is achieved by several tools, including Rsync (which we compare here), Vdelta and Xdelta.

III. Overview of Hermes

Figure 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 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 Figure 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.

IV. Byte level comparison

Hermes uses Zephyr's approach for byte level comparison to generate the delta script. For the sake of completeness, here we

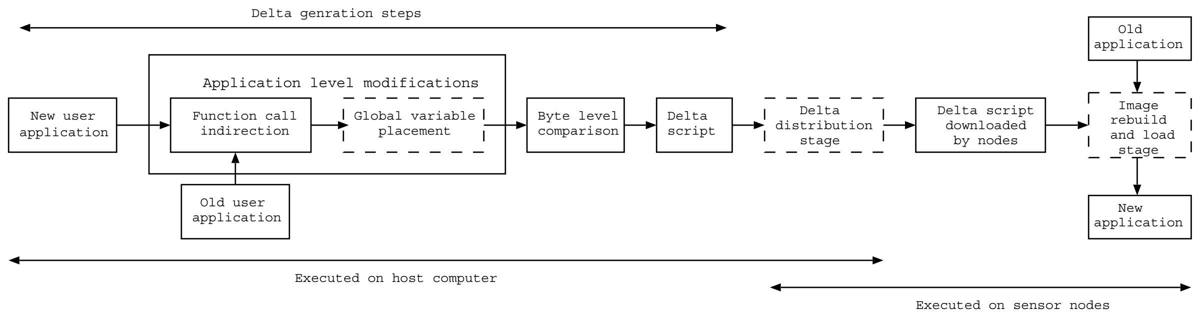

Fig. 1. Overview of Hermes: The stages with dashed rectangles are the ones which are introduced or modified by Hermes.

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 2183 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.

V. 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 Figure 2-a be changed to the one shown in Figure 2-b 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 Figure 2-c (for old image) and Figure 2-d (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 (x=1,2,...,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 2183 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.

A. 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 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 Figure 3-a, 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.

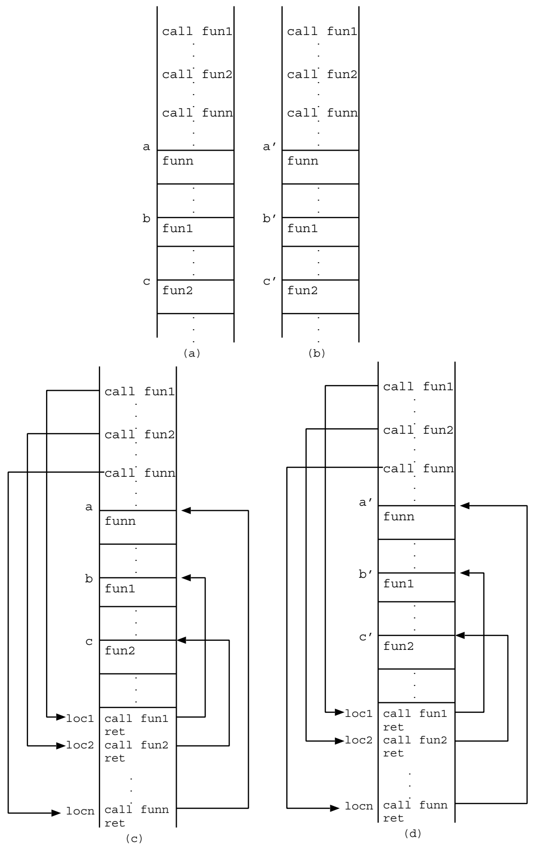

Fig. 2. (a) Old and (b) new images without indirection table (current state). Note that positions of the functions (fun1, etc.) have changed leading to changes in the call instructions; (c) Old and (d) new images with indirection table in Hermes. Note that due to the indirection table, the call instructions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

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 6090 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.

B. 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 software. These would correspond to variables which were present in the old software, but not in the new software. If there are empty slots, 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. For example, let default RAM structures for old and new applications be as shown in Figure 3-a and Figure 3-b respectively. The old application has initialized global variables $iv_1, iv_2, ..., iv_n$ in the .data section and uninitialized global variables $uv_1, uv_2, ..., uv_n$ 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 (Figure 3-b). As a result, global variables $iv_2, iv_3, ..., iv_n, uv_1, uv_2, ...uv_n$ 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 Figure 3-c 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 VIII, we discuss how Hermes avoids this gap. In the new application (Figure 3-d), 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

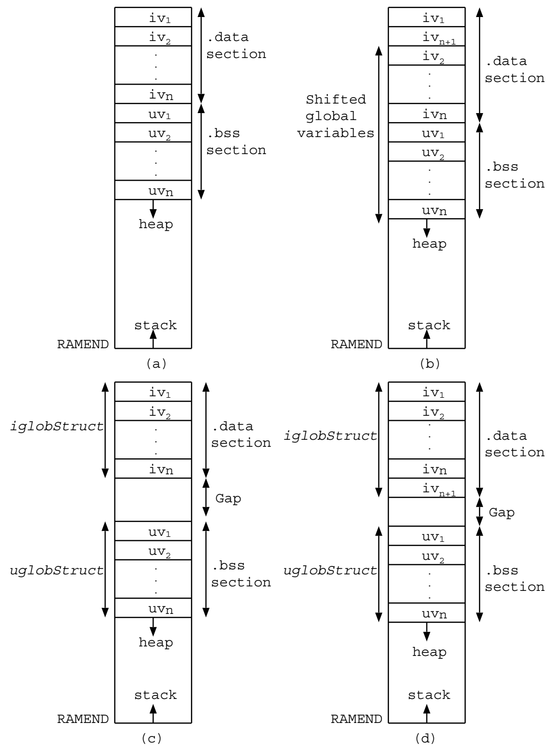

Fig. 3. Baseline RAM structures for (a) old and (b) new applications. RAM structures for corresponding (c) old and (d) new applications using Hermes.

resulting in a small delta script.

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 6090 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.

VI. Delta distribution stage

For wirelessly distributing the delta script, Hermes uses the approach from Stream [8] 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, 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 Figure 4. The reprogramming component and delta script are stored as image 0 and image 1 respectively. Image 2 and image 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 VII. 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 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 3-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.

VII. 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 (Figure 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 indi-

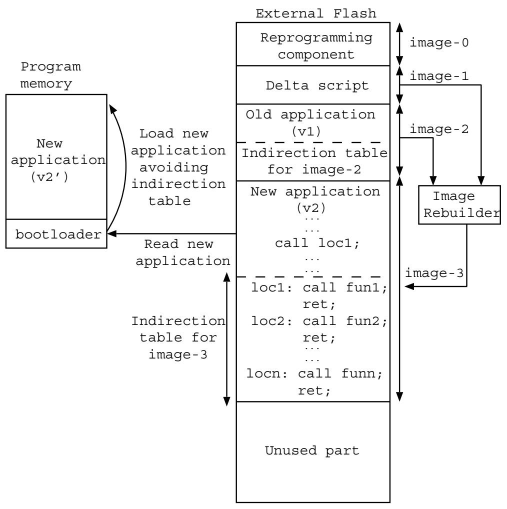

Fig. 4. Image rebuild and load stage. The right side shows the structure of external flash in Hermes.

rections to mitigate the effects of the function shifts. Use of one extra level of indirection increases the latency of the user program. Though it might look like one such indirection increases the time taken for one function call by only few clock cycles (e.g. 8 clock cycles for the AVR platform), it should be noted that the increase in latency accumulates over time. This is especially true for sensor networks where applications typically run in a loop — sample the sensor, process the sensed data, send data to some sink node, and then repeat the same process. Many functions are called in each iteration of the loop and the latency increases over time.

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 Figure 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 Figure 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 Figure 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.

VIII. 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 4KB RAM. Next we explain how we solve this problem in Hermes.

One possible soultion 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, respectively for Von-Neumann (e.g. msp430 platform [16]) and Harvard (e.g. AVR platform [6]) architectures. 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 [17] (msp430 platform) sensor nodes. Note that program memory is larger than RAM on the sensor nodes (e.g. TMote has 10KB RAM and 48KB program memory). Reprogramming protocol that we use occupies only about 25KB of program memory and hence enough space is available for .bss variables in program memory.

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 [18] 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 4KB RAM.

IX. 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 [19] 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 [18] 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.

A. *Size of delta script*

Table I shows the ratios of the number of bytes required to be transmitted for reprogramming by Deluge, Stream, Rsync and Zephyr to 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. Deluge, Stream, Rsync and Zephyr take up to 201, 134, 64 and 62 times more bytes than Hermes, respectively. Koshy and Pandey [14] 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 [14] would be identical to that of Hermes to Zephyr. Table I shows that [14] 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 [9] 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 [9] approach needs to transfer 3.14 to 64.75 times more bytes than Hermes.

TABLE I
Comparion of number of bytes to be transmitted by various approaches

	Deluge:Hermes	Stream:Hermes	Rsync:Hermes	Zephyr:Hermes	Hermes
Case 1	148.62	84.92	63.47	39.04	156
Case 2	34.81	19.89	12.49	4.11	666
Case 3	12.37	7.66	5.64	2.73	1874
Case 4	13.41	8.3	6.14	2.95	1729
Case A	13.52	9.01	5.96	1.79	1960
Case B	15.21	10.14	6.62	1.96	1742
Case C	5.5	3.8	3.14	2.08	5223
Case D	45.65	30.43	26.02	15.51	653
Case E	201.41	134.27	64.75	62.09	148

B. *Testbed experiments*

We perform testbed experiments using Mica2 [18] 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,

TABLE II
Ratio of reprogramming times of other approaches to Hermes

	Deluge:Hermes			Stream:Hermes			Rsync:Hermes			Zephyr:Hermes		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Case 1	24.77	44.66	34.24	14.12	25.96	20	10.98	19.78	15.63	7.69	16.08	11.39
Case 2	19.02	50.67	30.16	10.62	29.45	17.8	7.66	19.21	12.27	2.25	5.71	3.6
Case 3	6.14	13.48	9.8	4.77	9.15	6.37	3.37	5.57	4.56	2.06	3.56	2.8
Case 4	6.13	13.55	10.37	4.78	9.2	6.74	3.38	6.54	4.87	1.97	3.72	2.94
Case A	6.58	14.95	11.36	4.98	10.41	7.8	3.66	6.67	5.13	1.62	2.84	2.06
Case B	7.07	15.39	11.95	5.35	10.65	8.21	3.87	7.09	5.33	1.64	2.59	2.05
Case C	3.95	6.2	4.92	2.69	4.14	3.32	2.27	3.23	2.88	1.73	2.31	2.01
Case D	26.83	76.61	45.21	18.09	44.78	27.77	16.22	40.81	25.61	8.99	22.91	14.67
Case E	36.97	78.16	59.23	23.9	47.83	36.81	21.05	42.8	29.51	13.56	25.83	17.92

TABLE III
Ratio of number of packets transmitted during reprogramming by other approaches to Hermes

	Deluge:Hermes			Stream:Hermes			Rsync:Hermes			Zephyr:Hermes		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Case 1	28.54	140.31	91.83	17.05	78.5	49.87	11.02	53.28	33.2	6.23	35.43	20.26
Case 2	13.84	60.72	31.73	8.42	34.84	17.45	4.75	19.27	9	3.26	11.57	5.72
Case 3	5.93	13.03	10.4	4.16	8.21	6.45	2.89	6.34	4.73	1.67	2.66	2.12
Case 4	6.2	13.26	10.11	4.04	7.84	6.27	2.6	5.96	4.59	1.77	2.53	2.12
Case A	6.34	14.79	11.56	4.51	10.7	7.88	3.03	6.64	5.11	1.86	2.28	2.02
Case B	6.37	16.53	12.41	4.53	11.46	8.46	3.03	7.71	5.49	1.85	2.26	2.01
Case C	3.94	7.6	6.17	2.84	6.02	4.4	2.49	4.74	3.68	1.56	2.85	2.3
Case D	18.87	103.12	46.34	12.64	49.1	27.7	11.63	46.74	24.91	6.94	30.27	14.63
Case E	46.67	194.19	124.29	26	114.93	76.91	20.65	87.28	59.27	12.54	53.18	35.12

TABLE IV
Simulation results: Ratio of reprogramming time and number of packets transmitted by other approaches to Hermes

	6x6		8x8		10x10		12x12		14x14	
	Time	# Pkts	Time	# Pkts	Time	# Pkts	Time	# Pkts	Time	# Pkts
Deluge:Hermes	27.41	61.11	53.61	60.07	70.87	73.02	76.88	105.68	94.3	149.82
Stream:Hermes	15.55	34.16	40.01	38.68	48.25	45.4	53.28	67.66	70.52	97.55
Rsync:Hermes	12.34	26.68	2.12	27.87	28.43	34.22	38.16	49.56	54.43	74.77
Zephyr:Hermes	8.31	17.33	10.69	16.81	13.93	21.22	22.73	29.96	34.27	46.28

occasionally a non-adjacent node will receive a packet. In our experiments, if a node receives a packet from a non-adjacent node, it is dropped. This kind of software topology control has been used in other works also [20], [21]. 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 the line is the base node for linear networks. We provide quantitative comparison of Hermes with Deluge [7], Stream [8], Rsync (Jeong and Culler [9]) and Zephyr [4]. Note that Jeong and Culler [9] reprogram only nodes within one hop of the base node, but we used their approach on top of multi hop reprogramming protocol to provide a fair comparison. We perform these experiments for grids of size 2x2 to 4x4 and linear networks of size 2 to 10 nodes. The results presented here are the minimum, maximum and average over these grid and linear networks.

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 [21] to measure the network reprogramming time. Table II compares the ratio of reprogramming times of other approaches to Hermes. As expected, Hermes outperforms the non incremental reprogramming protocols Deluge and Stream significantly. Hermes is also 2.27 to 42.8 times faster than Rsync [9]. 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 [14] is same as that of Zephyr. Hermes is 1.62 to 25.83 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 4x4 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. Table III 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 (1.56 to 53.18 times) in energy.

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 [18], processes the sampled data 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. Figure 5 shows the additional latency due to indirections in all function calls during the work period. That is, the amount of time taken by Zephyr is larger than that by Hermes by the amount shown in Figure 5. By removing the indirection table, Hermes saves this latency, enabling lower duty cycle. So the nodes can sleep for this extra time and hence the amount of energy saved is significant in the long run.

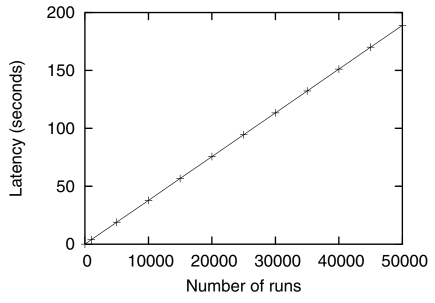

Fig. 5. Execution latency due to indirection table

C. Simulation Results

We perform TOSSIM [22] simulations on grid networks of varying size (up to 14x14) to demonstrate the scalability of Hermes and to compare it with other schemes. Table IV 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 3-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.

X. Conclusions

In this paper, we presented a multi-hop incremental reprogramming protocol 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 as well as to incrementally upload the changed modules in operating systems like SOS and Contiki that provide the dynamic linking feature. 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.
[2] C. Han, R. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava, "SOS: A dynamic operating system for sensor networks," *Mobisys*, pp. 163–176, 2005.
[3] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," *IEEE Emnets*, pp. 455–462, 2004.
[4] [Online]. Available: http://www.ece.purdue.edu/~dcsl/publications/papers/2008/zephyr_TR0808.%pdf
[5] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," *PhD thesis, Australian National University*, 1999.
[6] http://www.atmel.com.
[7] J. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," *SenSys*, pp. 81–94, 2004.
[8] R. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," *IEEE Infocom*, 2007.
[9] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," *IEEE SECON*, pp. 25–33, 2004.
[10] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *ACM SIGOPS Operating Systems Review*, pp. 85–95, 2002.
[11] J. Koshy and R. Pandey, "VMSTAR: synthesizing scalable runtime environments for sensor networks," *SenSys*, pp. 243–254, 2005.
[12] P. Levis, D. Gay, and D. Culler, "Active sensor networks," *2nd USENIX/ACM NSDI*, 2005.
[13] P. Rickenbach and R. Wattenhofer, "Decoding code on a sensor node," *DCOSS*, pp. 400–414, 2008.
[14] J. Koshy and R. Pandey, "Remote incremental linking for energy-efficient reprogramming of sensor networks," *EWSN*, pp. 354–365.
[15] P. Marron, M. Gauger, A. Lachenmann, O. Minder, D.and Saukh, and K. Rothermel, "FLEXCUP: A flexible and efficient code update mechanism for sensor networks," *EWSN*, pp. 212–227, 2006.
[16] http://www.ti.com/msp430.
[17] http://www.sentilla.com.
[18] http://www.xbow.com.
[19] http://estadium.purdue.edu.
[20] A. Kamra, V. Misra, J. Feldman, and D. Rubenstein, "Growth codes: maximizing sensor network data persistence," *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 255–266, 2006.
[21] R. Panta, I. Khalil, S. Bagchi, and L. Montestruque, "Single versus Multi-hop Wireless Reprogramming in Sensor Networks," *TridentCom*, pp. 1–7, 2008.
[22] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: accurate and scalable simulation of entire tinyOS applications," *SenSys*, pp. 126–137, 2003.