Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation

Rajesh Krishna Panta, Saurabh Bagchi, Samuel P. Midkiff
School of Electrical and Computer Engineering, Purdue University
{*rpanta,sbagchi,smidkiff*}*@purdue.edu*

Abstract

Wireless reprogramming of sensor nodes is an essential requirement for long-lived networks since the software functionality changes over time. The amount of information that needs to be wirelessly transmitted during reprogramming should be minimized since reprogramming time and energy depend chiefly on the amount of radio transmissions. In this paper, we present a multi-hop incremental reprogramming protocol called Zephyr that transfers the *delta* between the old and the new software and lets the sensor nodes rebuild the new software using the received delta and the old software. It reduces the delta size by using application-level modifications to mitigate the effects of function shifts. Then it compares the binary images at the byte-level with a novel method to create small delta, that is then sent over the wireless network to all the nodes. For a wide range of software change cases that we experimented with, we find that Zephyr transfers 1.83 to 1987 times less traffic through the network than Deluge, the standard reprogramming protocol for TinyOS, and 1.14 to 49 times less than an existing incremental reprogramming protocol by Jeong and Culler.

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 situated in places which are difficult to reach physically. 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, the reprogramming protocol should minimize the amount of information that needs to be wirelessly transmitted during reprogramming.

In practice, software running on a node evolves, with incremental changes to functionality, or modification of the parameters that control current functionality. Thus 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 challenges. A class of operating systems, including 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 [6] and Contiki [5], do support dynamic linking. However, their reprogramming support also does not handle 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 for runtime linking increases the amount of traffic

that needs to be disseminated through the network.

In this paper, we present a fully functional incremental multi-hop reprogramming protocol called *Zephyr*. It transfers the changes to the code, does not need dynamic linking on the node and does not transfer symbol and relocation tables. Zephyr uses an optimized version of the Rsync algorithm [20] to perform *byte-level comparison* between the old and the new code binaries. However, even an optimized difference computation at the low level can generate large deltas because of the change in the positions of some application components. Therefore, before performing byte-level comparison, Zephyr performs *application-level modifications*, most important of which is function call indirections, to mitigate the effects of the function shifts caused by software modification.

We implement Zephyr on TinyOS and demonstrate it on real multi-hop networks of Mica2 [2] nodes and through simulations. Zephyr can also be used with SOS or Contiki to upload incremental changes within a module. We evaluate Zephyr for a wide range of software change cases,—from a small parameter change to almost complete application rewrite—, using applications from the TinyOS distribution and various versions of a real world sensor network application called eStadium [3] deployed at the Ross-Ade football stadium at Purdue University. Our experiments show that Deluge [7], Stream [16], and the incremental protocol by Jeong and Culler [8] need to transfer up to 1987, 1324, and 49 times more number of bytes than Zephyr, respectively. This translates to a proportional reduction in reprogramming time and energy for Zephyr. Furthermore, Zephyr enhances the robustness of the reprogramming process in the presence of failing nodes and lossy or intermittent radio links typical in sensor network deployments due to significantly smaller amount of data that it needs to transfer across the network.

Our contributions in this paper are as follows: 1) We create a small-sized delta for dissemination using optimized byte-level comparisons. 2) We design application-level modifications to increase the structural similarity between different software versions, also leading to small delta. 3) We allow modification to any part of the software (kernel plus user code), without requiring dynamic linking on sensor nodes. 4) We present the design, implementation and demonstration of a fully functional multi-hop reprogramming system. Most previous work has concentrated on some of the stages of the incremental reprogramming system, but not delivered a functional complete system.

2 Related work

The question of reconfigurability of sensor networks has been an important theme in the community. Systems such as Mate [10] and ASVM [11] 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. Zephyr 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. Several protocols provide reprogramming with full binaries, such as Deluge [7] and Stream [16]. For incremental reprogramming, Jeong and Culler [8] 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 use any application-level modifications to handle function shifts. We compare the delta size generated by their approach and use it with an existing multi-hop reprogramming protocol to compare their reprogramming time and energy with Zephyr. In [19], the authors 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. Koshy and Pandey [9] use 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 of Flexcup [13] 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 component along with the new symbol and relocation tables to 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.

Reconfigurability is simplified in OSes like SOS [6] and Contiki [5]. In these systems, individual modules can be loaded dynamically on the nodes. Some modules can be quite large and Zephyr enables the upload of only the changed portions of a module. Specific challenges exist in the matter of reconfiguration in individual systems. SOS uses position independent code and due to architectural limitations on common embedded platforms, the relative jumps can only be within a certain offset (such as 4 KB for the Atmel 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 [9]). Zephyr, while currently implemented in TinyOS, can also support incremental reprogramming in these OSes by enabling incremental

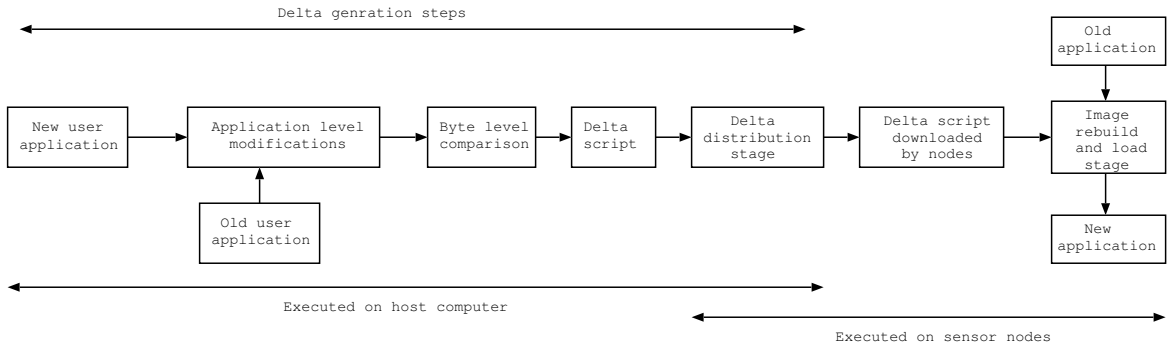


Figure 1: Overview of Zephyr

updates to changed module and updates to kernel modules.

Distinct from this work, in [15], we show that further orthogonal optimizations are possible to reduce the delta size, e.g., by mitigating the effect of shifts of global data variables. One of the drawbacks of Zephyr is that the latency due to function call indirection increases linearly with time. This is especially true for sensor networks because typical sensor applications operate in a loop — sample the sensor, perform some computations, transmit/forward the sensed value to other nodes and repeat the same process. In [15], we solve this while loading the newly rebuilt image from the external flash to the program memory by replacing each jump to the indirection table with a call to the actual function by reading the function address from the indirection table. In this way, we can completely avoid the function call latency introduced by Zephyr.

3 High level overview of Zephyr

Figure 1 is the schematic diagram showing various stages of Zephyr. First Zephyr performs application-level modifications on the old and new versions of the software to mitigate the effect of function shifts so that the similarity between the two versions of the software is increased. Then the two executables are compared at the byte-level using a novel algorithm derived from the Rsync algorithm [4]. This produces the delta script which describes the difference between the old and new versions of the software. These computations are performed on the host computer. The delta script is transmitted wirelessly to all the nodes in the network using the delta distribution stage. In this stage, first the delta script is injected by the host computer to the base node (a node physically attached to the host computer via, say a serial port). The base node then wirelessly sends the delta script to all nodes in the network, in a multi hop manner, if required. The nodes save the delta script in their external flash memory. After the sensor nodes complete downloading the delta script, they rebuild the new image using the delta and the old image and store it in the external flash. Finally the bootloader loads the newly built image from the external flash to the program memory and the node runs the new software. We describe these stages in the following sections. We first describe byte-level comparison and show why it is not sufficient and thus motivate the need for application-level modifications.

4 Byte-level comparison

We first describe the Rsync algorithm [20] and then our extensions to reduce the size of the delta script that needs to be disseminated.

4.1 Application of Rsync algorithm

Rsync is an algorithm originally developed to update binary data between computers over a low bandwidth network. Rsync divides the files containing the binary data into fixed size blocks. Both sender and receiver compute the pair (Checksum, MD4) over each block. If this algorithm is used as is for incremental reprogramming, then the sensor nodes need to perform expensive MD4 computations for the blocks of the binary image that they have. So, we modify Rsync such that all the expensive operations regarding delta script generation are performed on the host computer and not on the sensor nodes. The modified algorithm runs on the host computer only and works as follows: 1) The algorithm first generates the pair (Checksum, MD4 hash) for each block of the old image and stores them in a hash table. 2) The checksum is calculated for the first block of the new image. 3) The algorithm checks if this checksum matches the checksum for any block in the old image by hash-table lookup. If a matching block is found, Rsync compares if their MD4 hash also match. If MD4 also matches, then that block is considered as a matching block. If no matching block is found for either checksum or MD4, then the algorithm moves to the next byte in the new image and the same process is repeated until a matching block is found. Note that if two blocks do not have the same checksum, then MD4 is not computed for that

block. This ensures that the expensive MD4 computation is done only when the inexpensive checksum matches between the 2 blocks. The probability of collision is not negligible for two blocks having the same checksum, but with MD4 the collision probability *is* negligible.

After running this algorithm, Zephyr generates a list of COPY and INSERT commands for matching and non matching blocks respectively:

```
COPY <oldOffset> <newOffset> <len>
INSERT <newOffset> <len> <data>
```

COPY command copies *len* number of bytes from *oldOffset* at the old image to *newOffset* at the new image. Note that *len* is equal to the block size used in the Rsync algorithm. INSERT command inserts *len* number of bytes, i.e. *data*, to *newOffset* of the new image. Note that this *len* is not necessarily equal to the block size or its multiple.

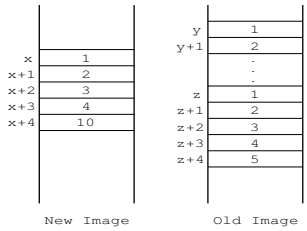


Figure 2: Finding super block

4.2 Rsync optimization

With the Rsync algorithm, if there are n contiguous blocks in the new image that match n contiguous blocks in the old image, n number of COPY commands are generated. We change the algorithm so that it finds the largest contiguous matching block between the two binary images. Note that this does not simply mean merging n COPY commands into one COPY command. As shown in Figure 2, let the blocks at the offsets x and $x+1$ in the new image match those at the offsets y and $y+1$ respectively in the old image. Let blocks at x through $x+3$ of the new image match those at z through $z+3$ respectively of the old image. Note that blocks at x and $x+1$ match those at y and $y+1$ and also at z and $z+1$. The Rsync algorithm creates two COPY commands as follows: COPY $\langle y \rangle$ $\langle B \rangle$ $\langle x \rangle$ and COPY $\langle y+1 \rangle$ $\langle B \rangle$ $\langle x+1 \rangle$, where B is the block size. Then simply combining these 2 commands as COPY $\langle y \rangle$ $\langle 2*B \rangle$ $\langle x \rangle$ does not result in the largest contiguous matching block. The blocks at the offsets z through $z+3$ form the largest contiguous matching block. We call contiguous matching blocks a *super-block* and the largest super-block the *maximal super-block*. The optimized Rsync algorithm finds the maximal super-block and uses that as the operand in the COPY command. Thus, optimized Rsync produces a single COPY command as COPY $\langle z \rangle$ $\langle 4*B \rangle$ $\langle x \rangle$. Figure 3 shows the pseudo code for optimized Rsync. Its complexity is $O(n^2)$ where n is the number of bytes in the image. This is not of a concern because the algorithm is run on the host computer and not on the sensor nodes and only when a new version of the software needs to be disseminated. As we will show in Section 8.2, optimized Rsync running on the desktop computer took less than 4.5 seconds for a wide range of software change cases that we experimented with.

```
/* Terminology
mbl=matching block list
cbl=contiguous block list
*/
1.   j=0 and cblStretch=0
2.   while j< number of bytes in the new image
3.     mbl=findAllMatchingBlocks(j)
4.     if mbl is empty
5.         j++
6.         if cbl is not empty
7.             Store any one element in cbl as maximum superblock
8.         go to 2
9.         else
10.          j=j+blockSize
11.          if (cblStretch==0)
12.            cbl=mbl
13.            cblStretch++
14.            go to 2
15.          else
16.            Empty tempCbl
17.            for each element in cbl do
18.              if (cbl.element + cblStretch == any entry in mbl )
19.                tempCbl=tempCbl U {cbl.element}
20.              if tempCbl is empty
21.                Store any one element in cbl as maximum superblock
22.                Empty cbl
23.                cblStretch=0
24.              else
25.                cbl=tempCbl
26.                cblStretch++
27.              go to 2
28. end while
findAllMatchingBlocks ( j )
     /*Same as Rsync algorithm, but instead of  returning the offset
     of just one matching block, returns a linked list consisting
     of offsets of all matching blocks in the old image for the
     block starting at offset j in the new image.*/
```

Figure 3: Pseudo code of optimized Rsync that finds maximal super block

4.3 Drawback of using only byte-level comparison

To see the drawback of using optimized Rsync alone, we consider two cases of software changes.

Case 1: Changing Blink application: Blink is an application in TinyOS distribution that blinks an LED on the sensor node every one second. We change the application from blinking green LED every second to blinking it every 2 seconds. Thus, this is an example of a small parameter change. The delta script produced with optimized Rsync is 23 bytes which is small and congruent with the actual amount of change made in the software.

Case 2: We added just 4 lines of code to Blink. The delta script between the Blink application 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 the previous case, but the delta script produced by optimized Rsync in this case is disproportionately larger.

When a single parameter is changed in the application as in Case 1, 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 2, the functions following those lines are shifted. As a result, all the calls to those functions refer to new locations. This produces several changes in the binary file resulting in the large delta script.

The boundaries between blocks can be defined by Rabin fingerprints as done in [18, 14]. A Rabin fingerprint is the polynomial representation of the data modulo a predetermined irreducible polynomial. These fingerprints are efficient to compute on a sliding window in a file. It should be noted that Rabin fingerprint can be a substitute for byte-level comparison only. Because of the content-based boundary between the chunks in Rabin fingerprint approach, the editing operations change only the chunks affected by those edits even if they change the offsets. Only the chunks that have changed need to be sent. But when the function addresses change, all the chunks containing the calls to those functions change and hence need to be sent explicitly. This results in a large delta—comparable to the delta produced by the optimized Rsync algorithm without application-level modifications. Also the *anchors* that define the boundary between the blocks have to be sent explicitly. The chunks in Rabin fingerprints are typically quite large (8 KB compared to less than 20 bytes for our case). As we can see from Figure 6, the size of the difference script will be much larger at 8 KB than at 20 bytes.

5 Application-level modifications

The delta script produced by comparison at the byte-level is not always consistent with the amount of change made in the software. This is a direct consequence of neglecting the application-level structures of the software. So we need to make modifications at the application-level so that the subsequent stage of byte-level comparison produces delta script congruent in size with the amount of software change. One way of tackling this problem is to leave some slop (empty) space after each function as in [9]. With this approach, even though a function expands (or shrinks), the location of the following functions will not change as long as the expansion is accommodated by the slop region assigned to that function. But this approach wastes program memory which is not desirable for memory-constrained sensor nodes. Also, this creates a host of complex management issues like what should be the size of the slop region (possibly different for different functions), what happens if the function expands beyond the assigned slop region, etc. Choosing too large a slop region means wastage of precious memory and too small a slop region means functions frequently need to be relocated. Another way of mitigating the effect of function shifts is by making the code position independent [6]. Position independent code (PIC) uses relative jumps instead of absolute jumps. However, not all architectures and compilers support this. For example, the AVR platform allows relative jumps within 4KB only and for MSP430(used in TelOS nodes), no compiler is known to fully support PIC.

5.1 Function call indirections

For the byte-level comparison to produce a small delta script, it is necessary to make the adjustments at the application-level to preserve maximum similarity between the two versions of the software. For example, let the application shown in Figure 4-a be changed such that the functions *fun1, fun2*, and *funn* are shifted from their original positions *b, c*, and *a* to new positions b', c', and a' respectively. Note that there can be (and generally will be) more than one call to a function. When these two images are compared at the byte-level, the delta script will be large because all the calls to these functions in the new image will have different target addresses from those in the old image. The approach we take to mitigate the ef-

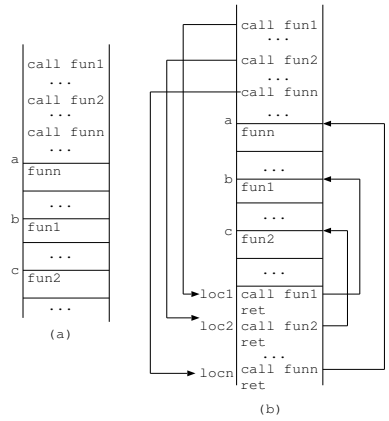


Figure 4: Program image (a) without indirection table and (b) with indirection table.

fects of function shifts is as follows: Let the application be as shown in Figure 4-a. We modify the linking stage of the executable generation process to produce the code as shown in Figure 4-b. Here calls to functions *fun1, fun2,..., funn* are replaced by jumps to *fixed locations loc1, loc2,..., locn* respectively. In common embedded platforms, the call can be to an arbitrarily far off location. The segment of the program memory starting at the fixed location *loc1* acts like an indirection table. In this table, the actual calls to the functions are made. When the call to the actual function returns, the indirection table directs the flow of the control back to the line following the call

to *loc-x (x=1,..., n)*. The location of the indirection table is kept fixed in the old and the new versions to reduce the size of the delta.

When the application shown in Figure 4-a is changed to the one where the functions *fun1, fun2,..., funn* are shifted, during the process of building the executable for the new image, we add the following features to the linking stage: When a call to a function is encountered, it checks if the indirection table in the old file contains the entry for that function (we also supply the old file (Figure 4-b) as an input to the executable generation process). If yes, then it creates an entry for that function at the indirection table in the new file at the same location as in the old file. Otherwise it makes a decision to assign a slot in the indirection table for that function (call it a *rootless* function) but does not yet create the slot. After assigning slots to the existing functions, it checks if there are any empty slots in the indirection table. These would correspond to functions which were called in the old file but are not in the new file. If there are empty slots, it assigns those slots to the rootless functions. If there are still some rootless functions without a slot, then the indirection table is expanded with new entries to accommodate these rootless function. Thus, the indirection table entries are naturally garbage collected and the table expands on an as-needed basis. 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, namely the 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.

This approach ensures that the segments of the code, except the indirection table, preserve the maximum similarity between the old and new images because the calls to the functions are redirected to the fixed locations even when the functions have moved in the code. The basic idea behind function call indirections is that the location of the indirection table is fixed and hence the target addresses of the jump to the table are identical in the old and new versions of the software. If we do not fix the location of the indirection table, the jump to indirection table will have different target addresses in the two versions of the software. As a result, the delta script will be large. In situations where the functions do not shift (as in Case 1 discussed in Section 4.3) Zephyr will not produce a delta script larger than optimized Rsync without indirection table. This is due to the fact that the indirection tables in the old and the new software match and hence Zephyr finds the large super-block that also contains the indirection table.

The linking changes in Zephyr are transparent to the user. She does not need to change the way she programs. The linking stage automatically makes the above modifications. Also Zephyr introduces one level of indirection during function calls, but the overhead of function call indirection is negligible because each such indirection takes only few clock cycles (e.g., 8 clock cycles on the AVR platform).

5.2 Pinning the interrupt service routines

It should be noted that due to the change in the software, not only the positions of the user functions but those of the interrupt service routines can also change. Such routines are not explicitly called by the user application. In most of the microcontrollers, there is an interrupt vector table at the beginning of the program memory (generally after the reset vector at 0x0000). Whenever an interrupt occurs, the control goes to the appropriate entry in the vector table that causes a jump to the required interrupt service routine. Zephyr does not change the interrupt vector table to direct the calls to the indirection table as explained above for the normal functions. Instead it modifies the linking stage to always put the interrupt service routines at fixed locations in the program memory so that the targets of the calls in the Interrupt vector table do not change. This further preserves the similarity between the versions of the software.

6 Metacommands for common patterns of changes

After the delta script is created through the above mentioned techniques, Zephyr scans through the script file to identify some common patterns and applies the following optimizations to further reduce the delta size.

6.1 CWI command

We noticed that in many cases, the delta script has the following sequence of commands:

```
COPY <oldOffset=O1> <len=L1> <newOffset=N1>
INSERT <newOffset> <len=l1> <data1>
COPY <oldOffset=O2> <len=L2> <newOffset=N2>
INSERT <newOffset> <len=l1> <data2>
```

and so on. Thus, small INSERT commands would be present in between large COPY commands, e.g., due to different operands *op* in instruction *ldi r24, op* commonly found in TinyOS programs while pushing *task* to the task queue. Here we have COPY commands that copy large chunks of size *L1, L2, L3, ...* from the old image followed by INSERT commands with very small values of *len= l1*. Further we notice that *O1+L1+l1=O2*, *O2+L2+l1=O3*, and so on. In other words, if the blocks corresponding to INSERT commands with small *len* had matched, we would have obtained a very large superblock. So we replace such sequences with the COPY_WITH_INSERTS (CWI) command.

```
CWI <oldOffset=O1> <newOffset=N1>
<len=L1+l1+...+Ln> <dataSize=l1>
<numInserts=n> <addr1> <data1>
<addr2> <data2> ... <addrn> <datan>
```

Here *dataSize=l1* is the size of *datai (i=1,2..., n), numInserts=n* is the number of *(addr,data)* pairs, *datai* are the data that have to be inserted in the new image at the offset *addri*. This command tells the sensor node to copy the *len=L1+l1+...+Ln* number of bytes of data from the old image at offset *O1* to the new image at the offset *N1*, but to insert *datai* at the offset *addri (i=1, 2, ..., n)*.

6.2 REPEAT command

This command is useful for reducing the number of bytes in the delta script that is used to transfer the indirection table. As shown in Figure 4-c and 4-d, the indirection table consists of the pattern *call fun1, ret, call fun2, ret , ...* where the same string of bytes (say *S1 = ret; call*) repeats with only addresses for *fun1, fun2*, etc. changing between them. So we use the following command to transfer the indirection table.

```
REPEAT <newOffset> <numRepeats=n>
<addr1> <addr2> ... <addrn>
```

This command puts the string *S1* at the offset *newOffset* in the new image followed by *addr1*, then *S1*, then *addr2*, and so on till *addrn*. Note that we could have used the CWI command for this case also. But since the string *S1* is fixed, we gain more advantage using the REPEAT command. This optimization is not applied if the addresses of the call instructions match in the indirection tables of the old and new images. In that case, COPY command is used to transfer the identical portions of the indirection table.

6.3 No offset specification

We note that if we build the new image on the sensor nodes in a *monotonic* order, then we do not need to specify the offset in the new file in any of the above commands. Monotonic means we always write at location x of the new image before writing at location y, for all $x < y$. Instead of the new offset, a counter is maintained and incremented as the new image is built and the next write always happens at this counter. So, we can drop the *newOffset* field from all the commands.

We find that for Case 2, where some functions were shifted due to addition of few lines in the software, the delta script produced with the application-level modifications is 280 bytes compared to 2183 bytes when optimized Rsync was used without application-level modifications. The size of the delta script without the meta-commands is 528 bytes. This illustrates the importance of application-level modifications in reducing the size of the delta script and making it consistent with the amount of actual change made in the software.

7 Delta distribution stage

One of the factors that we considered for the delta distribution stage was to have as small a delta script as possible even in the worst case when there is a huge change in the software. In such a case there is very little similarity between the old and the new code images and the delta script basically consists of a large INSERT command to insert almost the entire binary image. To have small delta script even in such extreme cases, it is necessary that the binary image itself be small. Since the binary image transmitted by Stream [16] is almost half compared to that of Deluge [7], Zephyr uses the approach from Stream with some modifications for wirelessly distributing the delta script. 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. Unlike Deluge, Stream does not transfer 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.

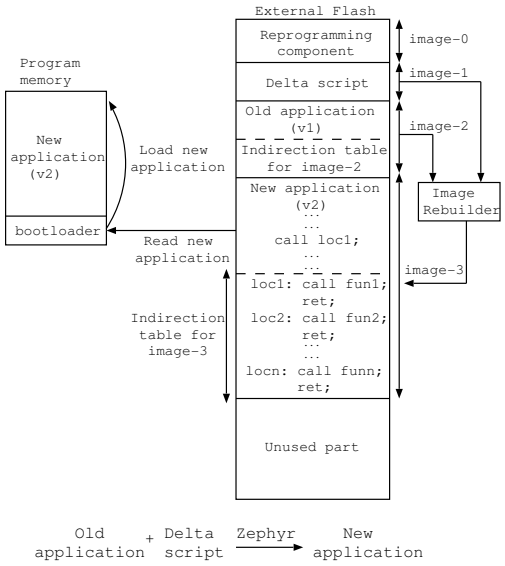


Figure 5: Image rebuild and load stage. The right side shows the structure of external flash in Zephyr.

Distinct from Stream, Zephyr divides the external flash as shown in the right side of Figure 5. 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 7.1. 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 to reboot all nodes in the network from image 0 (i.e. the 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, Zephyr transfers the delta script only.

6) All nodes store the received delta script as image 1.

7.1 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, INSERT, CWI, and REPEAT commands of the delta script and creates the new image which is stored as image 3 in the external flash.

The methods of rebooting from the new image are slightly different in Stream and Zephyr. In Stream, a node automatically reboots from the new code once it has completed the code update and it has satisfied all other nodes that depend on this node to download the new code. This means that different nodes in the network start running the new version of the code at different times. However, for Zephyr, we modified Stream so that all the nodes reboot from the new code after the user manually sends the reboot command from the base station (as in Deluge). We made this change because in many software change cases, the size of the delta script is so small that a node (say n_1) nearer to the base station quickly completes downloading the code before a node (say n_2) further away from the base station even starts requesting packets from n_1. As a result, n_1 reboots from the new code so fast that n_2 cannot even start the download process. Note that this however does not pose a correctness issue. After n_1 reboots from the new code, it will switch again to the reprogramming state when it receives advertisement from n_2. However, this incurs the performance penalty of rebooting from a new image. Our design choice has a good consequence —all nodes come up with the new version of the software at the same time. This avoids the situation where different nodes in the network run different versions of the software. When a node receives the reboot command , its bootloader loads the new software from image 3 of the external flash to the program memory (Figure 5). In the next round of reprogramming, image 3 will become the old image and the newly rebuilt image will be stored as image 2. As we will show in Section 8.3, the time to rebuild the image is negligible compared to the total reprogramming time.

7.2 Dynamic page size

Stream divides the binary image into fixed-sized pages. The remaining space in the last page is padded with all 0s. Each page consists of 1104 bytes (48 packets per page with 23 bytes payload in each packet). With Zephyr, it is likely that in many cases, the size of the delta script will be much smaller than 1104 bytes. For example, we have delta script of sizes of 17 bytes and 280 bytes for Case 1 and Case 2 respectively. Also, as we will show in Section 8.2, during the natural evolution of the software, it is more likely that the nature of the changes will be small or moderate and as a result, delta scripts will be much smaller than the standard page size. After all, the basic idea behind any incremental reprogramming protocol is based on the assumption that in practice, the software changes are generally small so that the similarities between the two versions of the software can be exploited to send only small delta. When the size of the delta script is much smaller than the page size, it is wasteful to transfer the whole page. So, we change the basic Stream protocol to use dynamic page sizes.

When the delta script is being injected to the base node, the host computer informs it of the delta script size. If it is less than the standard page size, the base node includes this information in the advertisement packets that it broadcasts. When other nodes receive the advertisement, they also include this information in the advertisement packets that they send. As a result, all nodes in the network know the size of the delta script and they make the page size equal to the actual delta script size. So unlike Deluge or Stream which transmit all 48 data packets per page, Zephyr transmits only required number of data packets if the delta script size is less than 1104 bytes. Note that the granularity of this scheme is the packet size, i.e., the last packet of the last page may be padded with zeros. But this results in small enough wastage that we did not feel justified in introducing the additional complexity of dynamic packet size. Our scheme can be further modified to advertise the actual number of packets

of the last page. This would minimize the wastage, for example in the case where the delta script has 1105 bytes, it would transfer 2 pages, the first page with 48 packets and the second with 1 packet.

8 Experiments and results

In order to evaluate the performance of Zephyr, we consider a number of software change scenarios. The software change cases for standard TinyOS applications that we consider are as follows:

Case 1: Blink application blinking a green LED every second to blinking every 2 seconds.

Case 2: Few lines added to the Blink application.

Case 3: Blink application to CntToLedsAndRfm: CntToLedsAndRfm is an application that displays the lowest 3 bits of the counting sequence on the LEDs as well as sends them over radio.

Case 4: CntToLeds to CntToLedsAndRfm: CntToLeds is the same as CntToLedsAndRfm except that the counting sequence is not transmitted over radio.

Case 5: Blink to CntToLeds.

Case 6: Blink to Surge: Surge is a multi hop routing protocol. This case corresponds to a complete change in the application.

Case 7: CntToRfm to CntToLedsAndRfm: CntToRfm is the same as CntToLedsAndRfm except that the counting sequence is not displayed on the LEDs.

In order to evaluate the performance of Zephyr with respect to natural evolution of the real world software, we considered a real world sensor network application called eStadium [3] deployed in Ross Ade football stadium at Purdue University. 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 [2] sensor board to one where few functions are added to sample the photo sensor also.

Case B: During the deployment phase, we decided to use opaque boxes for the sensor nodes. So, a few functions were deleted to remove the light sampling features.

Case C: In addition to temperature and battery voltage, we added the features for sampling all the sensors on the MTS310 board except light (e.g., microphone, accelerometer, magnetometer). This was a huge change in the software with the addition of many functions. For accelerometer and microphone, we collected mean and mean square values of the samples taken during a user specified window size.

Case D: This is the same as Case C but with addition of few lines of code to get microphone peak value over the user-specified window size.

Case E: We decided to remove the feature of sensing and wirelessly transmitting to the base node, the microphone mean value since we were interested in the energy of the sound which is given by the mean square value. A few lines of code were deleted for this change.

Case F: This is same as Case E except we added the feature of allowing the user to put the nodes to sleep for a user-specified duration. This was also a huge change in the software.

Case G: We changed the microphone gain parameter. This is a simple parameter change.

We can group the above changes into 4 classes:

Class 1 (Small change SC): This includes Case 1 and Case G where only a parameter of the application was changed.

Class 2 (Moderate change MC): This includes Case 2, Case D, and Case E. They consist of addition or deletion of few lines of the code.

Class 3 (Large change LC): This includes Case 5, Case 7, Case A, and Case B where few functions are added or deleted or changed.

Class 4 (Very large change VLC) : This includes Case 3, Case 4, Case 6, Case C, and Case F.

Many of the above cases involve changes in the OS kernel as well. In TinyOS, strictly speaking, there is no separation between the OS kernel and the application. The two are compiled as one big monolithic image that is run on the sensor nodes. So, if the application is modified such that new OS components are added or existing components are removed, then the delta generated would include OS updates as well. For example, in Case C, we change the application that samples additemperature and battery voltage to the one that samples microphone, magnetometer and accelerometer sensors in addition to temperature and battery. This causes new OS components to be added—the device drivers for the added sensors.

8.1 Block size for byte-level comparison

We modified Jarsync [4], a java implementation of the Rsync algorithm, to achieve the optimizations mentioned in Section 4.2. From here onward, by semi-optimized Rsync, we mean the scheme that combines two or more contiguous matching blocks into one super-block. It does not necessarily produce the maximal super-block. By optimized Rsync we mean our scheme that produces the maximal super-block but without the application-level modifications.

As shown in Figure 6, the size of the delta script produced by Rsync as well as optimized Rsync depends on the block size used in the algorithm. Recollect that the comparison is done at the granularity of a block. As ex-

Table 1: Comparison of delta script size of various approaches. Deluge, Stream and Rsync represent prior work.

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case A	Case B	Case C	Case D	Case E	Case F	Case G
Deluge:Zephyr	1400.82	85.05	4.52	4.29	8.47	1.83	29.76	7.60	7.76	2.63	203.57	243.25	2.75	1987.2
Stream:Zephyr	779.29	47.31	2.80	2.65	4.84	1.28	18.42	5.06	5.17	1.82	140.93	168.40	1.83	1324.8
Rsync:Zephyr	35.88	20.81	2.06	1.96	3.03	1.14	8.34	3.35	3.38	1.50	36.03	42.03	1.50	49.6
SemiOptimizedRsync:Zephyr	6.47	11.75	1.80	1.72	2.22	1.11	5.61	2.66	2.71	1.39	14.368	17.66	1.36	6.06
OptimizedRsync:Zephyr	1.35	7.79	1.64	1.57	2.08	1.07	3.87	2.37	2.37	1.35	7.84	9.016	1.33	1.4
ZephyrWithoutMetacommands:Zephyr	1.35	1.99	1.38	1.30	1.39	1.05	1.52	1.6	1.61	1.16	2.33	2.43	1.18	1.4

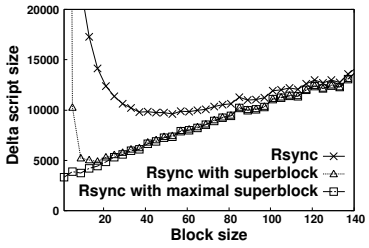
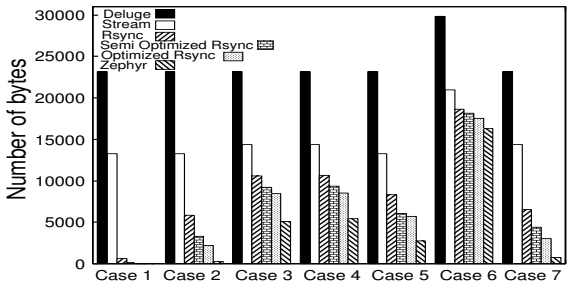
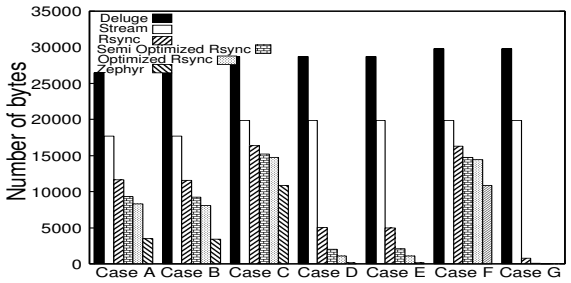


Figure 6: Delta script size versus block size

pected, Figure 6 shows that the size of the delta script is largest for Rsync and smallest for optimized Rsync. It also shows that as block size increases, the size of the delta script produced by Rsync and semi-optimized Rsync decreases till a certain point after which it has an increasing trend. The size of the delta script depends on two factors: 1) number of commands in the delta script and 2) size of data in the INSERT command. For Rsync and semi-optimized Rsync, for block size below the minima point, the number of commands is high because these schemes find lots of matching blocks but not (necessarily) the maximal super-block. As block size increases in this region, the number of matching blocks and hence the number of commands drops sharply causing the delta script size to decrease. However, as the block size increases beyond the minima point, the decrease in the number of commands in the delta script is dominated by the increase in the size of new data to be inserted. As a result, the delta script size increases. For optimized Rsync, there is a monotonic increasing trend for the delta script size as block size increases. There are however some small oscillations in the curve, as a result of which the optimal block size is not always one byte. The small oscillations are due to the fact that increasing the block size decreases the size of maximal superblocks and increases the size of data in INSERT commands. But sometimes the small increase in size of data can contribute to reducing the size of the delta script by reducing the number of COPY commands. Nonetheless, there is an overall increasing trend for optimized Rsync. This has the important consequence that a system administrator using Zephyr does not have to figure out the block size to use in uploading code for each application change. She can use the smallest or close to smallest block size and let Zephyr be responsible for compacting the size of the delta script. In all further experiments, we use the block size that gives the smallest delta script for each scheme—Rsync, semi-optimized Rsync, and optimized Rsync.



(a) TinyOS software change cases



(b) eStadium software change cases

Figure 7: Size of data transmitted for reprogramming

8.2 Size of delta script

The goal of an incremental reprogramming system is to reduce the size of the delta script that needs to be transmitted to the sensor nodes. 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. Figure 7 and Table 1 compare the delta script produced by Deluge, Stream, Rsync, semi-optimized Rsync, Optimized Rsync, and Zephyr. 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 the delta script. Deluge, Stream, Rsync, and semi optimized Rsync take up to 1987, 1324, 49, and 6 times more bytes than Zephyr, respectively. Note that for cases belonging to moderate or large change, the application

level modifications of Zephyr contribute to reducing the size of delta script significantly compared to optimized Rsync. Optimized Rsync takes up to 9 times more bytes than Zephyr. These cases correspond to shifts of some functions in the software. As a result, application-level modifications have great effect in those cases. In practice, these are probably the most frequently occurring categories of changes in the software. Case 1 and Case G are parameter change cases which do not shift any function. As a result, we find that delta scripts produced by optimized Rsync without application-level modifications are only slightly larger than the ones produced by Zephyr. Also even for very large software change cases (like cases 6, F, and C) Zephyr is more efficient compared to other schemes. In summary, application-level modifications have the greatest effects in moderate and large software change cases, significant effect in very large software change case (in terms of absolute delta size reduction) and small effect on very small software change cases.

Comparison with other incremental approaches: Rsync represents the algorithm used by Jeong and Culler [8] to generate the delta by comparing the two executables without any application-level modifications. We find that [8] produces up to 49 times larger delta script than Zephyr. Rsync also corresponds approximately to the system in [19] because it also compares the two executables without any application-level modifications. Koshy and Pandey [13] use a slop region after each function to minimize the likelihood of function shifts. Hence the delta script for their best case (i.e. when none of the functions expands beyond its slop region) will be same as that of Zephyr. But even in their best case scenario, the program memory is fragmented and less efficiently used than in Zephyr. This wastage of memory is not desirable for memory-constrained sensor nodes. When the functions *do* expand beyond the allocated slop region, they need to be relocated and as a result, all calls to those functions need to be patched with the new function addresses giving larger delta script than in Zephyr. Flexcup [13], though capable of incremental linking and loading on TinyOS, generates high traffic through the network due to large sizes of symbol and relocation tables. Also, Flexcup is implemented only on an emulator whereas Zephyr runs on the real sensor node hardware.

In the software change cases that we considered, the time to compile, link (with the application-level modifications) and generate the executable file was at most 2.85 seconds and the time to generate the delta script using optimized Rsync was at most 4.12 seconds on a 1.86 GHz Pentium processor. These times are negligible compared to the time to reprogram the network, for any but the smallest of networks. Further these times can be made smaller by using more powerful server-class machines. TinyOS applies extensive optimizations on the application binaries to run it efficiently on the resource-constrained sensor nodes. One of these optimizations involves inlining of several (small) functions. We do not change any of these optimizations. In systems which do not inline functions as TinyOS, Zephyr's advantage will be even greater since there will be more function calls. Zephyr's advantage will be minimum if the software change does not shift any function. For such a change, the advantage will be only due to the optimized Rsync algorithm. But such software changes are very rare, e.g. when only the values of the parameters in the program are changed. Any addition/deletion/modification of the source code in any function except the one which is placed at the end of the binary will cause the following functions to be shifted.

8.3 Testbed experiments

We perform testbed experiments using Mica2 [2] nodes for grid and linear topologies. For the grid network, the transmission range R_{tx} of a node is set such that $\sqrt{2}d < R_{tx} < 2d$, 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 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 non-adjacent node will receive a packet. In our experiments, if a node receives a packet from a non-adjacent node, it is dropped, to achieve a truly multi-hop network. A node situated at one corner of the grid or end of the line acts as the base node. We provide quantitative comparison of Zephyr with Deluge [7], Stream [16], Rsync [8] and optimized Rsync without application-level modifications. Note that Jeong and Culler [8] reprogram only nodes within one hop of the base node, but we used their approach on top of a multi hop reprogramming protocol to provide a fair comparison. The metrics for comparison are reprogramming time and energy. We perform these experiments for grids of size 2x2 to 4x4 and linear networks of size 2 to 10 nodes. We choose four software change cases, one from each equivalence class: Case 1 for Class 1 (SC), Case D for Class 2 (MC), Case 7 for Class 3 (LC), and Case C for Class 4 (VLC). Note that in the evaluations that follow, Rsync refers to the approach by Jeong and Culler [8].

8.3.1 Reprogramming time

Time to reprogram the network is the sum of the time to download the delta script and the time to rebuild the new image. Time to download the delta script is the time interval between the instant t_0 when the base node sends the first advertisement packet to the instant t_1 when the last node (the one which takes the longest time to download the delta script) completes downloading the delta script. Since clocks maintained by the nodes in the net-

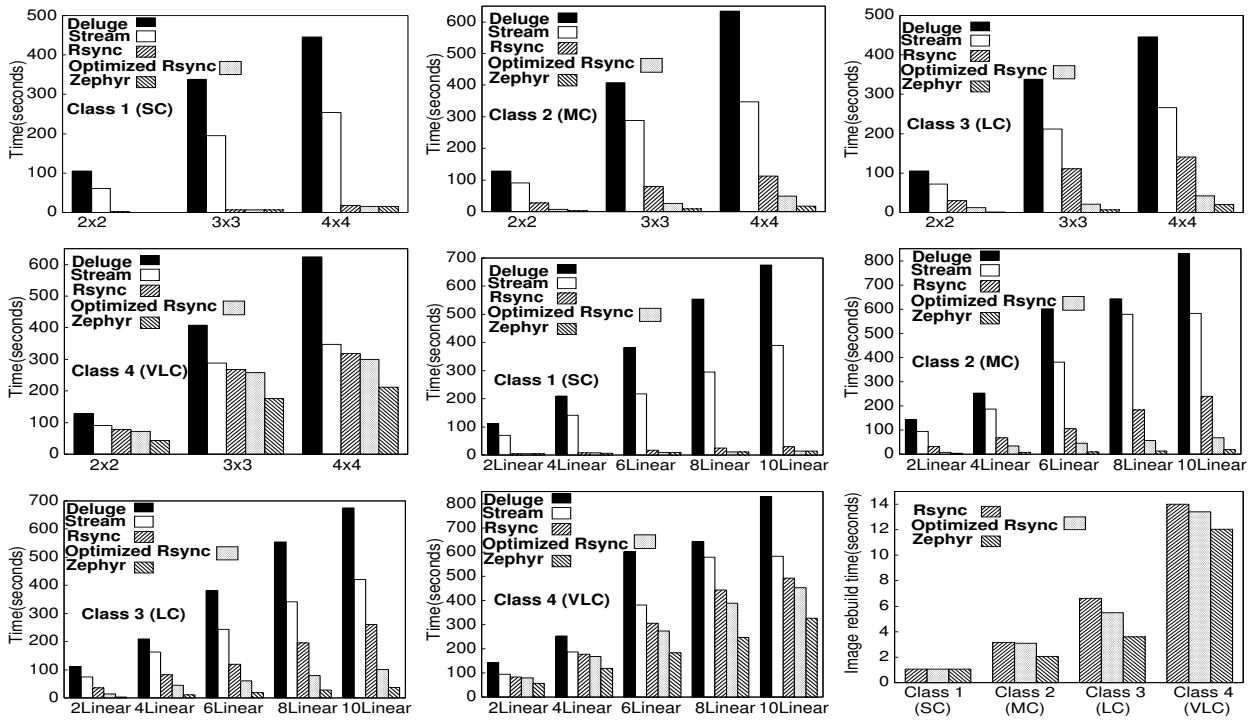


Figure 8: Comparison of reprogramming times for grid and linear networks. The last graph shows the time to rebuild the image on the sensor node.

Table 2: Ratio of reprogramming times of other approaches to Zephyr

	Class 1(SC)			Class 2(MC)			Class 3(LC)			Class 4(VLC)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Deluge:Zephyr	22.39	48.9	32.25	25.04	48.7	30.79	14.89	33.24	17.42	1.92	3.08	2.1
Stream:Zephyr	14.06	27.84	22.13	16.77	40.1	22.92	10.26	20.86	10.88	1.54	2.23	1.46
Rsync:Zephyr	1.03	8.17	2.55	5.66	12.78	8.07	5.22	10.89	6.50	1.34	1.71	1.42
Optimized Rsync:Zephyr	1.01	1.1	1.03	2.01	4.09	2.71	2.05	3.55	2.54	1.27	1.55	1.35

work are not synchronized, we cannot take the difference between the time instant t_1 measured by the last node and t_0 measured by the base node. To solve this synchronization problem, we use the approach of [17], which achieves this with minimal overhead traffic.

Figure 8 (all except the last graph) compares reprogramming times of other approaches with Zephyr for different grid and linear networks. Table 2 compares the ratio of reprogramming times of other approaches to Zephyr. It shows minimum, maximum and average ratios over these grid and linear networks. As expected, Zephyr outperforms non-incremental reprogramming protocols like Deluge and Stream significantly for all the cases. Zephyr is also up to 12.78 times faster than Rsync, the approach by Jeong and Culler [8]. This illustrates that the Rsync optimization and the application-level modifications of Zephyr are important in reducing the time to reprogram the network. Zephyr is also significantly faster than optimized Rsync without application-level modifications for moderate, large, and very large software changes. In these cases, the software change causes the function shifts. So, these results show that application level modifications greatly mitigate the effect of function shifts and reduces the reprogramming time significantly. For small change case where there are no function shifts, Zephyr, as expected, is only marginally faster than optimized Rsync without application-level modifications. In this case, the size of the delta script is very small (17 and 23 bytes for Zephyr and optimized Rsync respectively) and hence there is not much to improve upon. Since Zephyr transfers less information at each hop, Zephyr's advantage will increase with the size of the network. The last graph in Figure 8 shows the time to rebuild the new image on a node. It increases with the increase in the scale of the software change, but is negligible compared to the total reprogramming time.

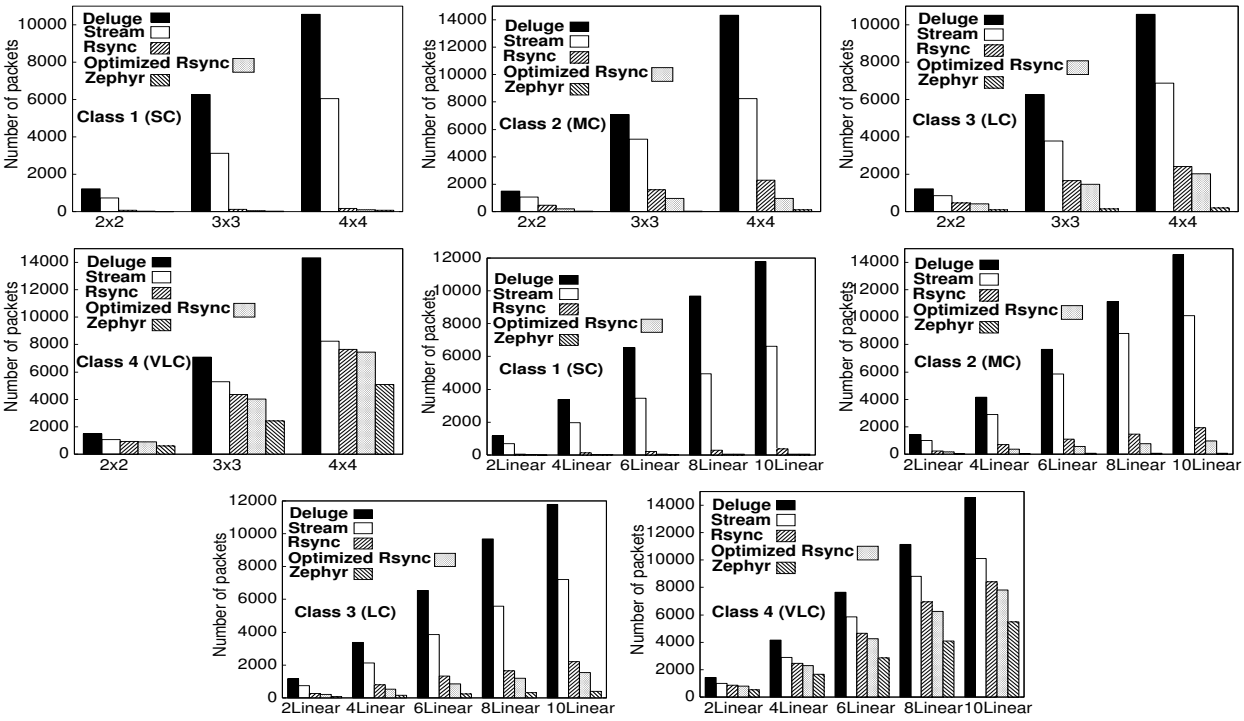


Figure 9: Comparison of number of packets transmitted during reprogramming.

Table 3: Ratio of number of packets transmitted during reprogramming by other approaches to Zephyr

	Class 1(SC)			Class 2(MC)			Class 3(LC)			Class 4(VLC)		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Deluge:Zephyr	90.01	215.39	162.56	40	204.3	101.12	12.27	55.46	25.65	2.51	2.9	2.35
Stream:Zephyr	53.76	117.92	74.63	28.16	146.1	82.57	8.6	36.19	15.97	1.62	2.17	1.7
Rsync:Zephyr	2.47	7.45	5.38	6.66	38.28	21.09	3.28	12.68	6.69	1.50	1.78	1.60
Optimized Rsync:Zephyr	1.13	1.69	1.3	4.38	22.97	9.47	2.72	10.58	3.95	1.38	1.64	1.49

8.3.2 Reprogramming energy

Among the various factors that contribute to the energy consumption 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. Figure 9 and Table 3 compare the number of packets transmitted by Zephyr with other schemes for grid and linear networks of different sizes. The number of bytes transmitted by all nodes in the network for reprogramming by Deluge, Stream, Rsync, and optimized Rsync is up to 215, 146, 38, and 22 times more than that by Zephyr. The fact that *Rsync:Zephyr>1* indicates that Zephyr is more energy efficient than the incremental reprogramming approach of [8]. The application-level modifications are significant in reducing the number of packets transmitted by Zephyr compared to optimized Rsync without such modifications. Note that in cases like Case 7 and Case D (moderate to large change class), application-level modifications have the greatest impact where the functions get shifted. Application-level modifications preserve maximum similarity between the two images in such cases thereby reducing the reprogramming traffic overhead. In cases where only some parameters of the software change without shifting any function, the application-level modifications achieve smaller reduction. But the size of the delta is already very small and hence reprogramming is not resource intensive in these cases. Even for very large software changes, Zephyr significantly reduces the reprogramming traffic.

8.4 Simulation Results

We perform TOSSIM [12] simulations on grid networks of varying size (up to 14x14) to demonstrate the scalability of Zephyr and to compare it with other schemes.

Figure 10 shows the reprogramming time and number of packets transmitted during reprogramming for Case D (Class 2 (MC)). We find that Zephyr is up to 92.9, 73.4, 16.1, and 6.3 times faster than Deluge, Stream, Rsync [8], and optimized Rsync without application-level modifications, respectively. Also, Deluge, Stream, Rsync [8], and optimized Rsync transmit up to 146.4, 97.9, 16.2, and 6.4 times more number of packets than Zephyr, respectively. Most software changes in practice are likely to belong to this class (moderate change) where we see that application-level modifications significantly reduce the reprogramming overhead. Zephyr inherits its scalability property from Deluge since none of the changes in Zephyr (except the dynamic page size) affects the network or is driven by the size 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.

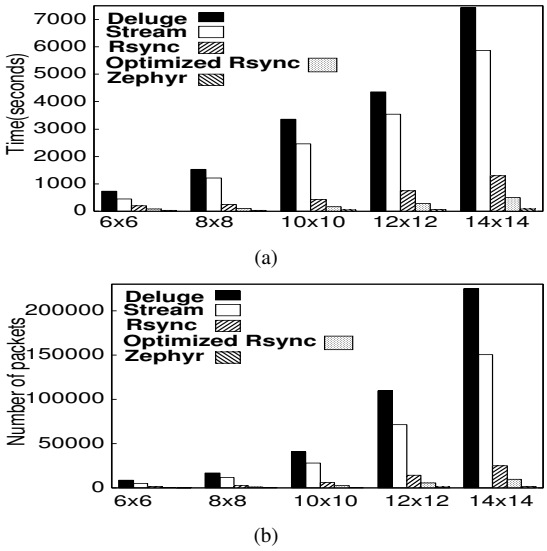


Figure 10: Simulation results for (a) reprogramming time and (b) number of packets transmitted during reprogramming (Case D, i.e. Class 2 (MC))

9 Conclusion

In this paper, we presented a multi-hop incremental reprogramming protocol called Zephyr 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 to use techniques like function call indirections to mitigate the effect of function shifts for 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. Our experimental results show that for a large variety of software change cases, Zephyr 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. We can also use multiple nodes as the source of the new code instead of a single base node to further speed up reprogramming.

References

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

[2] http://www.xbow.com.

[3] http://estadium.purdue.edu.

[4] http://jarsync.sourceforge.net/.

[5] DUNKELS, A., GRONVALL, B., AND VOIGT, T. Contiki-a lightweight and flexible operating system for tiny networked sensors. *IEEE Emnets* (2004), 455–462.

[6] HAN, C., RENGASWAMY, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. SOS: A dynamic operating system for sensor networks. *MobiSys* (2005), 163–176.

[7] HUI, J., AND CULLER, D. The dynamic behavior of a data dissemination protocol for network programming at scale. *SenSys* (2004), 81–94.

[8] JEONG, J., AND CULLER, D. Incremental network programming for wireless sensors. *IEEE SECON* (2004), 25–33.

[9] KOSHY, J., AND PANDEY, R. Remote incremental linking for energy-efficient reprogramming of sensor networks. *EWSN* (2005), 354–365.

[10] LEVIS, P., AND CULLER, D. Maté: a tiny virtual machine for sensor networks. *ACM SIGOPS Operating Systems Review* (2002), 85–95.

[11] LEVIS, P., GAY, D., AND CULLER, D. Active sensor networks. *NSDI* (2005).

[12] LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. TOSSIM: accurate and scalable simulation of entire tinyOS applications. *SenSys* (2003), 126–137.

[13] MARRON, P., GAUGER, M., LACHENMANN, A., MINDER, D.AND SAUKH, O., AND ROTHERMEL, K. FLEXCUP: A flexible and efficient code update mechanism for sensor networks. *EWSN* (2006), 212–227.

[14] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. *SOSP* (2001), 174–187.

[15] PANTA, R., AND BAGCHI, S. Hermes: Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks. *To appear in IEEE Infocom* (2009).

[16] PANTA, R., KHALIL, I., AND BAGCHI, S. Stream: Low Overhead Wireless Reprogramming for Sensor Networks. *IEEE Infocom* (2007), 928–936.

[17] PANTA, R., KHALIL, I., BAGCHI, S., AND MONTESTRUQUE, L. Single versus Multi-hop Wireless Reprogramming in Sensor Networks. *TridentCom* (2008), 1–7.

[18] PUCHA, H., ANDERSEN, D., AND KAMINSKY, M. Exploiting similarity for multi-source downloads using file handprints. *NSDI* (2007).

[19] REIJERS, N., AND LANGENDOEN, K. Efficient code distribution in wireless sensor networks. *WSNA* (2003), 60–67.

[20] TRIDGELL, A. Efficient Algorithms for Sorting and Synchronization, PhD thesis, Australian National University.