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

ABSTRACT

Wireless reprogramming of sensor nodes is an essential 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 *Zephyr*. It reduces the difference between the old and the new code images by performing *application level modifications*. Then it compares the binary images at the byte level with a method to create small *delta* that needs to be sent over the wireless network to all the nodes. Zephyr does not require dynamic linking and relocation on the sensor nodes and hence can be used on top of operating systems like TinyOS. It can also be applied for small changes within a "module" on operating systems like SOS and Contiki that provide dynamic linking. For a wide range of software change scenarios ranging from small parameter change to complete overhaul of the software, we find that our protocol achieves 45% to 99% reduction in the size of the information to be transmitted compared to Deluge. Our testbed experiments on mica2 motes show that Zephyr is 48.7 (for small parameter change) to 1.92 (for huge application change) times faster than Deluge for reprogramming networks. Also, Deluge transmits 215.39 to 2.51 times more packets than Zephyr for the software modification cases that we experimented with. We also show the scalability of Zephyr using TOSSIM simulations.

Categories and Subject Descriptors

C.2.1 Network Architecture and Design and C.2.3 Network operations

General Terms

Design, Experimentation, Measurement, Performance

Keywords: Network reprogramming; Deluge; Rsync; Function call indirections.

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. The change 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. It is essential that the code update be 100% reliable and reach all the nodes that it is destined for. It is important to minimize the resource cost of the reprogramming, most significantly, the energy and the time spent in disseminating the code through the network.

In practice, the software running on a node evolves, with incremental changes to functionality, or modification of the parameters that control current functionality. The change may involve addition, deletion, or modification of one or a few components. 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. In incremental reprogramming, only the changes to the code are transmitted wirelessly and the entire code is reassembled at the node from the existing code and the received changes.

The design of incremental reprogramming on sensor nodes poses several practical challenges. A class of operating systems, that includes the widely used TinyOS [4], does not support dynamic linking of software components on a node. It executes on each node a single monolithic image, that contains what would be called the kernel parts in a traditional OS, as well as the application parts. 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 [8] and Contiki [7], 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 (PIC) 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 can increase the amount of traffic that needs to be disseminated through the network.

In this paper, we present a system called *Zephyr* that supports fully functional incremental multi-hop reprogramming in a sensor network. It transfers the changes to the code, does not need dynamic linking on the node and does not transfer symbol and relocation tables. Dynamic linking on the nodes can violate the low overhead requirement of the sensor nodes because linker requires

considerable computation resources and memory. Also, wireless transmission of symbol, relocation tables and other data structures for dynamic linking can consume significant energy. Zephyr uses byte level comparisons between existing and current code, building on the Rsync algorithm [24]. It optimizes the size of the difference that needs to be transferred (also referred to as the *delta*) by considering largest matching contiguous block of code. However, we observe that even an optimized difference computation at the low level can generate large deltas in cases where the positions of some application components change. Therefore, Zephyr performs high level modifications of the code at the application level such that function call instructions do not change. This requires a level of indirection for function calls but significantly reduces the size of delta where applications have high function reuse.

We implement Zephyr on TinyOS and demonstrate it on real multi-hop testbeds that are of grid and linear structures. Zephyr can also be used with SOS or Contiki to upload incremental changes within a module. We use some applications (and versions thereof) from the TinyOS distribution and different versions of a sensor network application that we have developed for sensing physical environment parameters and capturing still images at a local football stadium. We consider cases from a small change to the application to an almost complete rewrite and quantify the performance of Zephyr with respect to existing reprogramming protocols Deluge [9] and Stream [20]. Our testbed experiments show that Zephyr is 48.7 (for small parameter change) to 1.92 (for huge application change) times faster than Deluge for reprogramming networks. Also, Deluge transmits 215.39 to 2.51 times more packets than our approach for the software modification cases that we experimented with. The scalability results on the TOSSIM simulator are also impressive – Deluge takes 146 and 92 times more energy and reprogramming time respectively than Zephyr.

Our contributions in this paper are as follows:

1) We modify the linking process to use function call indirections to preserve maximum similarity between the two versions of the software so that the byte level comparison gives a small delta.

2) We create a new delta-computation algorithm based on Rsync suited to incremental reprogramming of sensor networks.

3) We allow modification of any part of the software, without requiring dynamic linking on the resource constrained 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, but not all.

The rest of the paper is structured as follows. Section 2 gives a brief overview of various stages of Zephyr. Section 3 discusses the byte level comparison and explains why such comparison alone is not sufficient. Section 4 presents the application level modifications. Section 5 explains more high level optimizations to reduce the delta size. Section 6 discusses the delta distribution method. Section 7 explains the testbed and the simulation setups and results. Section 8 surveys the related work and Section 9 concludes the paper.

2. HIGH LEVEL OVERVIEW OF ZEPHYR

Zephyr uses 3 stages to reprogram the network.

1) *Delta generation stage*: The computations for this stage are carried out on the host computer. The difference between the old and the new versions of the software is generated by performing application level modifications followed by comparison of the two versions at the byte level using the optimized Rsync algorithm. The difference basically consists of the commands to copy the specified region of the old image to the specified offset at the new image and to insert a string of bytes to the specified offset at the new image. We call this sequence of commands the *regenerate script* or *delta script*. Note that these are not virtual machine instructions as in [14, 16, 17] and the user does not have to program using virtual machine instructions. The regenerate script provides the necessary information to generate the new code image at the node.

2) *Delta distribution stage*: The delta script is injected by the host computer to the *base node*. A base node is the sensor 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.

3) *Image rebuild and load stage*: 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 first provide a description of the byte level comparison and show that the size of the delta script produced by the byte level comparison alone is not congruent with the amount of change made in the software. That is, even if a small amount of change is made in the software, depending on the nature of the change, it may result in a large delta script if only byte level comparison is used.

3. BYTE LEVEL COMPARISON

We first describe the Rsync algorithm [24] and then our extensions to reduce the size of the delta script that needs to be disseminated.

3.1 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 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. Let *B* be the block size used for the algorithm. 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.

3.2 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 1, 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

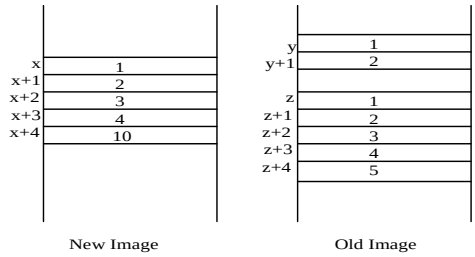


Figure 1: Finding super block

```
/*
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 2: Pseudo code of optimized Rsync that finds maximal super-block.

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` <*y*> <*B*> <*x*> and `COPY` <*y+1*> <*B*> <*x+1*>. Then simply combining these 2 commands as `COPY` <*y*> <*2*B*> <*x*> 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` <*z*> <*4*B*> <*x*>. Figure 2 shows the pseudo code for optimized Rsync. Its complexity is $O(n^2)$ where *n* is the number of bytes in the image. As we will show in Section 7.2, optimized Rsync running on the desktop computer took less than 4.5 seconds for a range of software change cases.

3.3 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 that comes with the standard TinyOS distribution. It blinks a LED 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 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, as was also noted in [13].

4. 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 has been done in [13]. 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 [8]. Position independent code (PIC) is defined by the use of 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, no compiler is known to fully support PIC.

4.1 Function call indirections

To make sure that the comparison at the byte level using the optimized Rsync algorithm results in 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. Let the application shown in Figure 3-a be changed to the one shown in Figure 3-b. Here 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 effects of function shifts is as follows: Let the application be as shown in Figure 3-a. We modify the linking stage of the executable generation process to produce the code as shown in Figure 3-c. 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.

When the application shown in Figure 3-a is changed to the one shown in Figure 3-b, 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 3-c) 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 for those functions in the new file that also existed in the old file, 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.

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. In situations where the functions do not shift (as in Case 1 that we discussed in Section 3.3) Zephyr will not produce a delta script larger than optimized Rsync without indirection table because the indirection tables in the old and the new software match and hence Zephyr finds the large super-block containing the indirection table also. Note that the function call indirection in Zephyr effectively creates slop regions—one before the indirection table and one after it. But this is clearly preferable to having slop regions after each function since here, the regions can be amortized over all the functions.

The linking changes in Zephyr are transparent to the user. She does not need to change the way she programs. She writes the application as in Figure 3-a and Figure 3-b. 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).

4.2 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

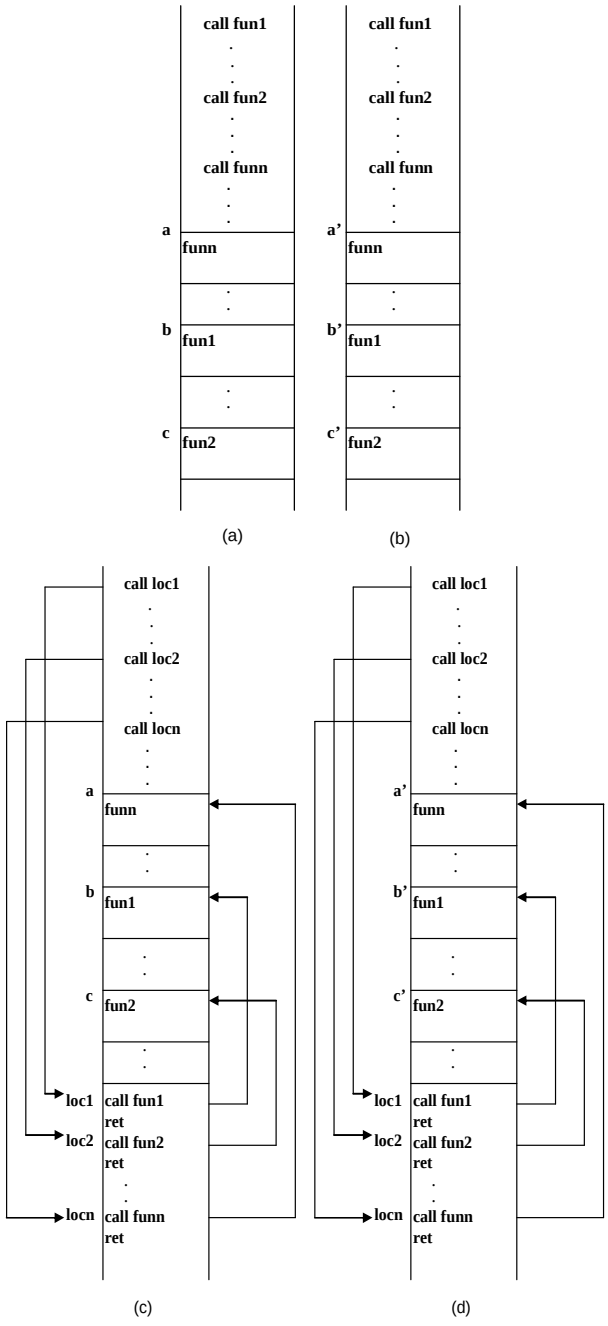


Figure 3: (a) Old image and (b) new image without application level modification; (c) Old image and (d) new image with application level modification.

fixed locations in the program memory so that the targets of the calls in the Interrupt vector table do not change. This

also helps to preserve the similarity between the versions of the software.

5. META COMMANDS 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.

5.1 CWI Command

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

`COPY <oldOffset=`O_1`> <len=`L_1`> <newOffset=`N_1`>`

`INSERT <newOffset> <len=`l_1`> <data`$_1$`>`

`COPY <oldOffset=`O_2`> <len=`L_2`> <newOffset=`N_2`>`

`INSERT <newOffset> <len=`l_1`> <data`$_2$`>`

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 AVR programs. Here we have COPY commands that copy large chunks of size L_1, L_2, L_3, … from the old image followed by INSERT commands with very small values of $len = l_1$. Further we notice that $O_1+L_1+l_1=O_2$ and $O_2+L_2+l_1=O_3$ 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 super-block. So we replace such sequences with the COPY_WITH_INSERTS (CWI) command.

`CWI <oldOffset=`O_1`> <len=`$L_1+l_1+\ldots+L_n$`>`
`<dataSize=`l_1`> <numInserts=n> <addr`$_1$`> <data`$_1$`>`
`<addr`$_2$`> <data`$_2$`> … <addr`$_n$`> <data`$_n$`>`

Here $dataSize=l_1$ is the size of $data_i$ (i=1,2…, n), $numInserts=n$ is the number of ($addr$, $data$) pairs, $data_i$ are the data that have to be inserted in the new image at the offset $addr_i$. This command tells to copy the $len=L_1+l_1+\ldots+L_n$ bytes of data from the old image at offset O_1 to the new image at the offset N_1, but to insert $data_i$ at the offset $addr_i$ (i=1, 2, …, n).

5.2 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 3-c and d, the indirection table consists of the pattern *call fun1, ret, call fun2, ret ,…* where the same string of bytes (say S_1 = *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> <addr`$_1$`>`
`<addr`$_2$`> … <addr`$_n$`>`

This command puts the string S_1 at the offset *newOffset* in the new image followed by $addr_1$, then S_1, then $addr_2$, and so on till $addr_n$. We could have used the CWI command for this case also. But since the string S_1 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 indirection table.

5.3 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 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.

With all these modifications, 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. This illustrates the importance of application level modifications in reducing the size of the delta script and making it consistent with the amount of change made in the software. The application level modifications capture the direct software changes instead of the indirect changes. These indirect changes occur due to the semantics of the instruction set and the application and are captured by byte level comparison.

6. DELTA DISTRIBUTION STAGE

For wirelessly distributing the delta script, we use the approach from Deluge [9] with some modifications. Deluge is the reprogramming protocol that comes with the standard TinyOS distribution. Zephyr uses the code distribution method of Deluge to transfer the delta script instead of transferring the entire software 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 protocol of advertisement, request, and code broadcast between neighboring nodes. One of the factors that we considered for the delta distribution method 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 case there is very little similarity between the old and the new 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.

Deluge attaches the user application with the entire Deluge reprogramming protocol code and sends them as one big image. To reduce the size of the delta script, we use the Stream protocol instead [20]. This protocol attaches a very small component to the user application instead of the whole reprogramming protocol. According to [20], Stream reduces the number of bytes to be transferred during reprogramming by almost half compared to Deluge. Next we describe Stream briefly.

6.1 Background on Stream

The reason behind Deluge's approach to attach the entire Deluge code with the user application is to make sure that Deluge is always running on the sensor nodes. This is done so that the nodes are receptive to future code updates since they are not capable of multitasking. Stream counters this problem by dividing the reprogramming component into 2 parts: Stream Reprogramming-Support (Stream-RS) and Stream Application Support (Stream-AS). Stream-RS is the core reprogramming component and is preinstalled on all the nodes, before deployment, as image 0 in the external flash (Figure 4(a)). Stream-AS is a small component that is attached to the user application and is stored as image 1 in the external flash. During the normal operation, all nodes in the network run image 1 (Stream-AS plus user application). When reprogramming is to be done, the user gives the command to the base node (node physically attached to the host computer) to reboot from image 0 (Stream-RS). The bootloader on the nodes load image 0 from the external flash to the program memory and all nodes in the network reboot from Stream-RS. Now the code distribution takes place using the usual 3-way handshake method of Deluge. When all nodes have downloaded the new application plus Stream-AS (stored as image 1 in external flash), the bootloader loads image 1 from the external flash to the program memory and all nodes reboot from the new image. In this way, Stream avoids transferring the code of the entire reprogramming protocol every time reprogramming is done.

6.2 Delta distribution protocol description

Zephyr divides the external flash as shown in Figure 4(b). The protocol works as follows: 1) Initially all nodes in the network are running image 2 (User application plus Stream-AS). At the host computer, delta script is generated between the old image (say image 2) and the new image. 2) The user gives the command to the base node to reboot all nodes in the network from image 0 (Stream-RS). 3) The base node broadcasts the reboot command and itself reboots from Stream-RS. 4) The nodes receiving the reboot command from the base node rebroadcast the reboot command and themselves reboot from Stream-RS. Note that this is a controlled flooding because each node broadcasts the reboot command only once and immediately reboots from Stream-RS. All nodes do the same and finally all nodes in the network run Stream-RS. 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. 6) All nodes receive the delta script and store it as image 1.

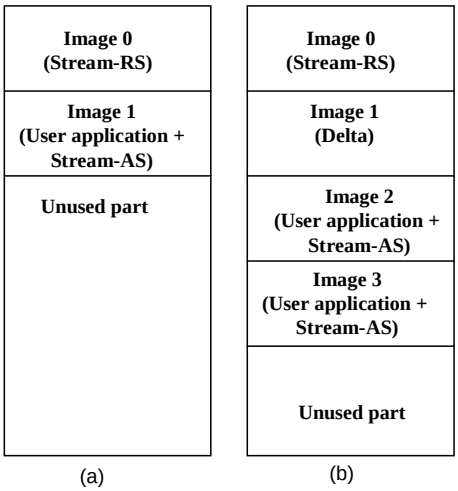


Figure 4: Structure of external flash in (a) Stream and (b) Zephyr

6.3 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). 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. In the next round of reprogramming, image 3 becomes the old image and the newly rebuilt image is stored as image 2. As we will show in Section 7, the time to rebuild the image is negligible compared to the total reprogramming time. Zephyr incurs higher flash use over Stream. The increase is given by the size of the user application plus Stream-AS plus the regenerate script.

6.4 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 7.2, during the natural evolution of the software, it is more likely that the nature of the changes to 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 (according to Deluge's algorithm). 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 page may be padded with zeroes. 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.

7. 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 considered are as follows:

Case 1: Blink application blinking 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 deployed at a football stadium to provide safety and security applications, infotainment applications 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, 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. 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 the 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.

7.1 Block Size for byte level comparison

We modified Jarsync [1], a java implementation of the Rsync algorithm, to achieve the optimizations mentioned

in Section 3.2. 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 applied before.

As shown in Figure 5, the size of the delta script produced by Rsync as well as optimized Rsync depends upon the block size used in the algorithm. Recollect that the comparison is done at the granularity of a block. As expected, Figure 5 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 regenerate 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 regenerate script is dominated by the increase in the size of new *data* to be inserted (i.e., the size of the INSERT commands). As a result, the delta script size increases.

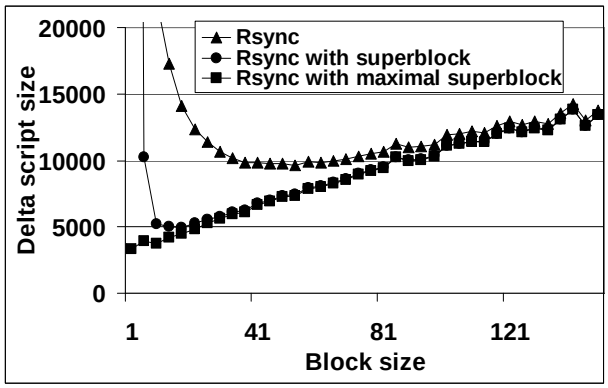


Figure 5: Delta script size versus block size

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 super-blocks 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 delta script. In all further experiments, we use the block size that gives the smallest delta script for each protocol.

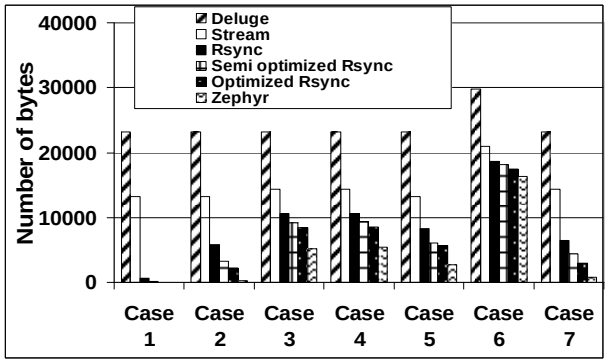
7.2 Size of regenerate script

Figure 6 (a) and (b) compare the size of the information to be transmitted for reprogramming in various software change cases mentioned above for Deluge, Stream, Rsync, semi optimized Rsync, Optimized Rsync and Zephyr protocol. 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. Table 1 shows the ratio of the number of bytes of delta script for other cases to Zephyr. We find that Zephyr significantly reduces the size of the delta script compared to other approaches. Deluge, Stream, Rsync, and semi optimized Rsync take upto 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 without application level modifications. 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. In such cases, Zephyr works best compared to optimized Rsync. This re-emphasizes the importance of the application level modifications. 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. This is because the application level modifications are meant to mitigate the effects of function shifts which do not happen in Case 1 and Case G. Also even for extreme software change cases (like cases 6, F and C) Zephyr is quite 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 (if we look at the actual size reduction) and small effect on very small software change cases. A small regenerate 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.

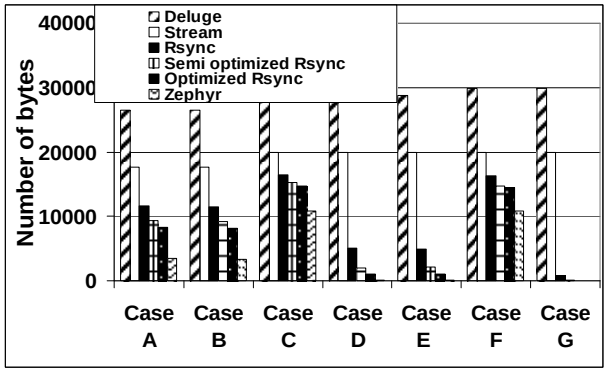
Table 1: Comparison of delta script size of Zephyr with other approaches

	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
SemiOptRsync/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
OptRsync/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

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.



(a)



(b)

Figure 6: Size of data transmitted for reprogramming

7.3 Testbed experiments

We perform the experiments using Mica2 [2] nodes. Testbed experiments are performed for 2 different network topologies: grid and linear. For each network topology, we define *neighbors* of a node n_1 as those nodes which can receive the packets sent by n_1. In our testbed experiments, if a node n_1 receives a packet from a node n_2 which is not its neighbor, the packet is dropped. In this way, we ensure that the network is truly multi hop. 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. For 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. This kind of software control has been used in other works also [12, 21]. We compare Zephyr with Deluge, Stream and optimized Rsync without application level modifications. The metrics for comparison are reprogramming time and energy. We choose 4 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).

7.3.1 Reprogramming time

Time to reprogram the network T is given by

$$T=T_D + T_R$$

where T_D is the time to download the delta script and T_R is 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 network 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 [21], which achieves this with minimal overhead traffic.

Figure 7 (except the last graph) compares reprogramming times of the four protocols 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. We see that Zephyr is much faster than Deluge and Stream for all the cases. Zephyr is also significantly faster than optimized Rsync without application level

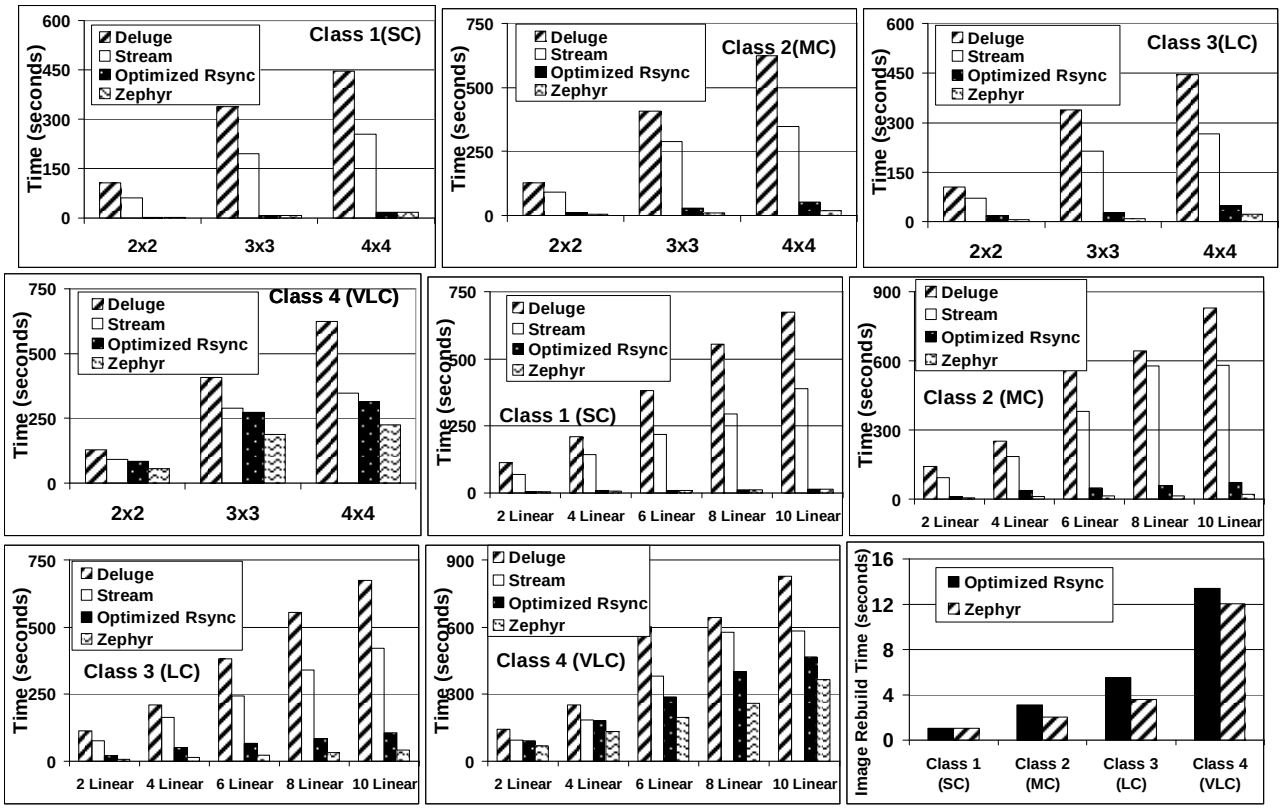


Figure 7: Comparison of reprogramming times for grid and linear networks. The last graph shows the time to rebuild the image on the sensor node.

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
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

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 modification greatly mitigates 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 bytes for Zephyr) and hence there is not much to improve upon. The result on reprogramming time is as predicted by the result on the size of the regenerate script from Section 7.2. The last graph in Figure 7 shows the time to rebuild the new image on a node. It increases with the increase in the size of delta script, but is negligible compared to the total reprogramming time.

7.3.2 Calculation of reprogramming energy

Among the various factors that contribute to the energy used in the process of 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 consumed. Figure 8 and Table 3 compare the number of packets transmitted by Zephyr with other schemes for grid and linear networks of different sizes. Like reprogramming time, Zephyr reduces the number of packets transmitted during reprogramming significantly compared to other approaches. The application level modifications are significant in reducing

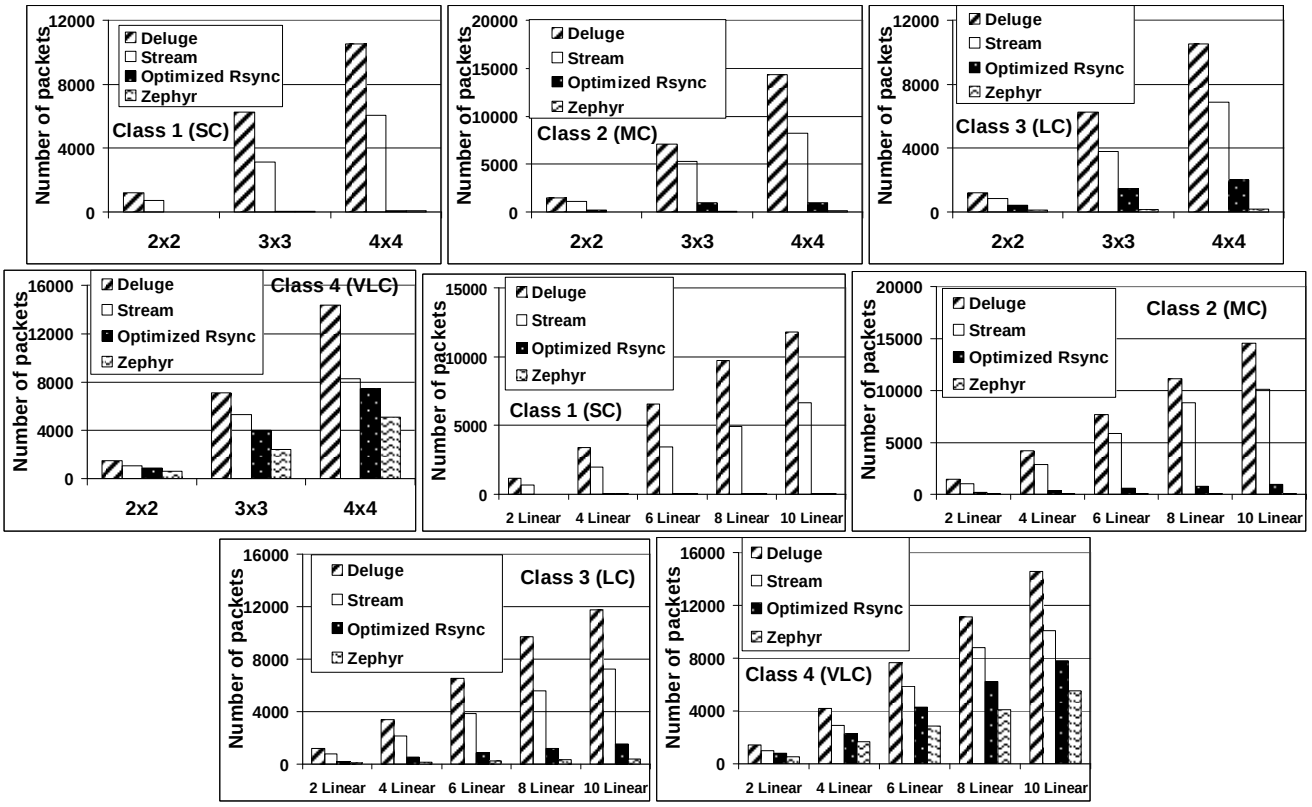


Figure 8 : Comparison of number of packets transmitted during reprogramming.

Table 3: Comparison of 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
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

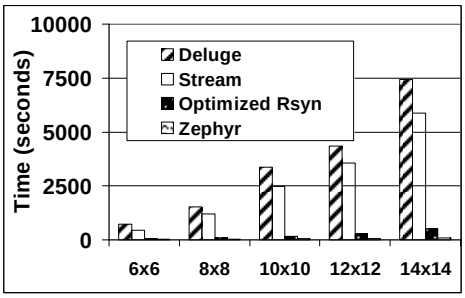
the number of packets transmitted by Zephyr compared to optimized Rsync without such modifications. It should be noted that in cases like 7 and D (moderate to large change class), application level modifications have the greatest impact where the functions get shifted. Application level modifications 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 modification achieves smaller reduction. But the size of the delta is already very small and hence this is not a problem. Even for very large software changes, Zephyr achieves significant reduction in the number of packet transmissions.

7.4 Simulation results

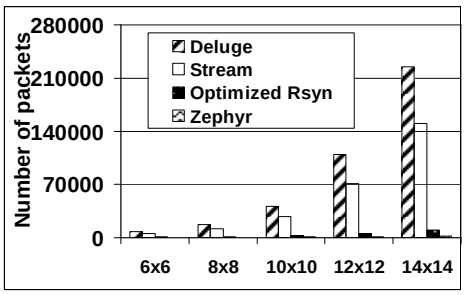
We perform TOSSIM [18] simulations to demonstrate the scalability of Zephyr and to compare it with other schemes. Since TOSSIM does not model the execution time accurately, the goal of these simulations is to exhibit the comparative performance and trend. Since it takes tens of hours to complete simulations for large networks, we reduce the number of packets per page from 48 to 24. This is not a serious concern because we are interested in comparing the various approaches and not on the absolute values. We use grid networks of varying size (up to 14x14). Figure 9 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, and 6.3 times faster than Deluge, Stream and optimized Rsync without application level modifications, respectively. Also, Deluge, Stream and optimized Rsync transmit up to 146.4, 97.9 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 is expected to be as scalable as 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.



(a)



(b)

Figure 9: Simulation results for (a) reprogramming time and (b) number of packets transmitted during reprogramming (Case D, i.e. Class 2 (MC))

8. RELATED WORK

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

Several systems, such as Maté [16], VM* [14], and ASVM [17] provide virtual machines that run on resource-constrained sensor nodes. They enable efficient code updates, since the code that runs on these virtual machines 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. There are several protocols that provide reprogramming with full binaries, such as Deluge and Stream. For incremental reprogramming, several researchers have computed deltas between existing and new images to send just the delta over the air to the nodes. In early work [11], the authors use Rsync to compute the difference. However, being built on top of XNP [10], it can only reprogram a single hop network. Also they compare only at the byte level and do not consider the application level structures. In [22], 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 [13] have the design goal of keeping address patches to a small number. They 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 in [19] 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, where the new image is created. 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.

Zephyr builds on existing work by providing efficient delta computation, basically enabling us to look at the differences at the finest granularity of an individual block. It combines this with application level code modifications to minimize the amount of traffic that needs to be disseminated. Importantly, previous work on incremental reprogramming has focused on one or some stages of the process while here we present the results of the complete multi-hop reprogramming process on a testbed.

Reconfigurability is simplified in OSes that support linkable modules. The prominent examples are SOS [8], Contiki [7], and RETOS [5]. In all of these, individual modules can be loaded. 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 protocols. SOS uses position independent code and due to architectural limitations on common embedded platforms, the relative jumps can be within a certain offset only (such as 4 KB for the Atmel AVR platform). Contiki disseminates the symbol and relocation tables, which may be quite large for cases where many functions are referenced in the

application. The function indirection approach of Zephyr has similarity with the RETOS way of implementing function calls from an application module to a kernel module. In it, the indirection is used to trigger runtime checks to protect the kernel from a buggy application. Zephyr, while currently implemented in TinyOS, can also support incremental reprogramming in these OSes by enabling incremental updates to an entire module and updates to kernel modules. In the latter case, Zephyr will have to run in a privileged mode.

There is a long history of dynamic linking in server and desktop applications [15] and with Java [3, 6, 23]. While reducing the volume of code is a concern, JavaSpec and QS [23] operate on bytecode with symbolic labels, and all of these operate on a classfile granularity, which encompasses multiple functions. Compilers for general purpose systems have the ability to generate position independent code, and do so to support dynamically linked shared libraries [15].

9. CONCLUSION

In this paper, we presented a multi-hop incremental reprogramming protocol called Zephyr that optimizes the reprogramming overhead by reducing the size of the regenerate script that needs to be disseminated through the network. To the best of our knowledge, the techniques like function call indirections that we use for application level modifications have not been used by any previous work 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 in operating systems like SOS and Contiki that do provide the dynamic linking feature. In the latter case, our scheme can be applied for transmitting the difference in the component(s) that has changed. Furthermore, Zephyr does not require resource constrained sensor nodes to be equipped with dynamic linking capabilities.

Like functions, global variables can also get shifted due to software changes. We can further reduce the size of delta by mitigating the effects of global data shifts. We leave that as our future work. Further, the issue of providing multiple points to inject code to speed up the reprogramming process is an open question.

10. REFERENCES

1. http://jarsync.sourceforge.net/.
2. http://www.xbow.com.
3. Arnold, K. and Gosling, J. *The Java programming language*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1998.
4. Berkeley, U.o.C. www.tinyos.net
5. Cha, H., Choi, S., Jung, I., Kim, H., Shin, H., Yoo, J. and Yoon, C. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN)*, ACM Press New York, NY, USA, 2007, 148-157.
6. Czajkowski, G. Application isolation in the Java Virtual Machine *OOPSLA*, ACM Press New York, NY, USA, 2000, 354-366.
7. Dunkels, A., Gronvall, B. and Voigt, T. Contiki-a lightweight and flexible operating system for tiny networked sensors *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, 2004, 455-462.
8. Han, C.C., Kumar, R., Shea, R., Kohler, E. and Srivastava, M., A Dynamic Operating System for Sensor Nodes. in *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, (2005).
9. Hui, J.W. and Culler, D., The dynamic behavior of a data dissemination protocol for network programming at scale. in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, (Baltimore, MD, USA, 2004), ACM Press, 81-94.
10. Inc., C.T. Mote In-Network Programming User Reference, 2003.
11. Jeong, J. and Culler, D. Incremental network programming for wireless sensors *Proceedings of the First IEEE Commnication Society Conference on Sensor and Ad Hoc Communications and Networks (SECON)*, 2004, 25-33.
12. Kamra, A., Misra, V., Feldman, J. and Rubenstein, D. Growth codes: maximizing sensor network data persistence *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, ACM Press New York, NY, USA, 2006, 255-266.
13. Koshy, J. and Pandey, R. Remote incremental linking for energy-efficient reprogramming of sensor networks *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005, 354-365.
14. Koshy, J. and Pandey, R. VMSTAR: synthesizing scalable runtime environments for sensor networks *Proceedings of the 3rd international conference on Embedded networked sensor systems (Sensys)*, ACM Press New York, NY, USA, 2005, 243-254.
15. Levine, J.R. *Linkers and Loaders*. Morgan Kaufmann, 1999.
16. Levis, P. and Culler, D. Mat\é: a tiny virtual machine for sensor networks. *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. 85-95.
17. Levis, P., Gay, D. and Culler, D. Active sensor networks *Proceedings of USENIX/ACM NSDI'05, Boston, MA, USA*, 2005.
18. Levis, P., Lee, N., Welsh, M. and Culler, D. TOSSIM: accurate and scalable simulation of entire tinyOS applications *Proceedings of the 1st international conference on Embedded networked sensor systems (Sensys)*, ACM Press New York, NY, USA, 2003, 126-137.
19. Marron, P.J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O. and Rothermel, K. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, Springer, 2006.
20. Panta, R.K., Khalil, I. and Bagchi, S. Stream: Low Overhead Wireless Reprogramming for Sensor Networks *IEEE Conference on Computer Communications (Infocom)*, 2007.
21. Panta, R.K., Khalil, I., Bagchi, S. and Montestruque, L. Single versus Multi-hop Wireless Reprogramming in Sensor Networks *4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities (TridentCom)*, 2008.
22. Reijers, N. and Langendoen, K. Efficient code distribution in wireless sensor networks *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, ACM Press New York, NY, USA, 2003, 60-67.
23. Serrano, M., Bordawekar, R., Midkiff, S. and Gupta, M. Quicksilver: a quasi-static compiler for Java *OOPSLA*, ACM Press New York, NY, USA, 2000, 66-82.
24. Tridgell, A. Efficient Algorithms for Sorting and Synchronization, Ph.D. Thesis, Australian National University, 1999.