Efficient Incremental Code Update for Sensor Networks

RAJESH KRISHNA PANTA, SAURABH BAGCHI, and SAMUEL P. MIDKIFF,
Purdue University

Wireless reprogramming of sensor nodes is an essential requirement for long-lived networks since software functionality needs to be changed over time. During reprogramming, the number of radio transmissions should be minimized, since reprogramming time and energy depend chiefly on the number of radio transmissions. In this article, we present a multihop incremental reprogramming protocol called Zephyr that transfers the *delta* between old and new software versions, and lets the sensor nodes rebuild the new software using the received delta and the old software. Zephyr reduces the delta size by using application-level modifications to mitigate the effects of function shifts. Then it compares the two binary images at the byte level to generate a small delta, that is then sent over the wireless network to all the nodes. For the wide range of software change cases that we used as benchmarks, Zephyr transfers 1.83 to 1987 times less traffic through the network than Deluge (the standard nonincremental reprogramming protocol for TinyOS) and 1.14 to 49 times less traffic than an existing incremental reprogramming protocol by Jeong and Culler [2004].

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Network communications; wireless communications*

General Terms: Design, Performance

Additional Key Words and Phrases: Network reprogramming, Rsync, Deluge

ACM Reference Format:
Panta, R. K., Bagchi, S., and Midkiff, S. P. 2011. Efficient incremental code update for sensor networks. ACM Trans. Sensor Netw. 7, 4, Article 30 (February 2011), 32 pages.
DOI = 10.1145/1921621.1921624 http://doi.acm.org/10.1145/1921621.1921624

1. INTRODUCTION

Large-scale sensor networks may be deployed for long periods of time. During this time the requirements from the network or the environment in which the nodes are deployed may change. This can require modifying the application executing on the sensor nodes, or providing the application with a different set of parameters. We will collectively refer to both of these changes as *reprogramming*. Once deployed, it may be very difficult to manually reprogram the sensor nodes because of the scale (possibly hundreds of nodes) and the embedded nature of the deployment, since the nodes may be located in places that are difficult to access physically. The most relevant form of reprogramming is *remote multihop reprogramming* using the wireless medium which reprograms the nodes as they are embedded in their sensing environment. Since the performance of the sensor network is degraded (possibly 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

Authors' addresses: R. K. Panta (corresponding author), S. Bagchi, S. P. Midkiff, Department of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN 47907; email: rpanta@purdue.edu.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 1550-4859/2011/02-ART30 $10.00
DOI 10.1145/1921621.1921624 http://doi.acm.org/10.1145/1921621.1921624

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 new application can be reassembled at the node from the existing application and the received changes. The goal of incremental reprogramming is to transfer a small delta, the difference between the old and the new software, so that reprogramming time and energy are minimized.

The design of incremental reprogramming protocol for sensor nodes poses several challenges. Many operating systems do not support dynamic linking of software components on a sensor node. For example, the standard release of TinyOS [tinyos], one of the widely used operating systems for sensor nodes, does not provide this feature. This rules out the straightforward transfer of only those components that have changed and dynamically linking them at the node. The second class of operating systems, represented by SOS [Han et al. 2005] and Contiki [Dunkels et al. 2004], do support dynamic linking. However, their reprogramming support also does not handle changes to the kernel modules. Moreover, the specifics of the position-independent code strategy employed in SOS limit 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 article, we present a fully functional incremental multihop 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 [Tridgell 1999] to perform *byte-level comparison* between the old and the new code binaries. As we will show, even an optimized difference computation at the low level generates large deltas because of changes in the position of application components. Therefore, before performing byte-level comparison, Zephyr performs *application-level modifications*, the most important of which is to use function call indirections to mitigate the effects of changes in the location of functions caused by software modification.

We implement Zephyr on TinyOS and demonstrate it using real multihop networks of Mica2 [xbow] 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 both the TinyOS distribution and various versions of a real-world sensor network application called eStadium [eStadium] that has been deployed at the Ross-Ade football stadium at Purdue University. Our experiments show that Deluge [Hui and Culler 2004], Stream [Panta et al. 2007], and the incremental protocol by Jeong and Culler [2004] 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 because of the significantly smaller amount of data that it needs to transfer across the network.

Our contributions in this article are as follows.

(1) We present a technique that uses optimized byte-level comparisons and leads to small deltas.

(2) We present application-level modifications that increase the structural similarity between different software versions, also leading to small delta.
(3) We present techniques that support modification of any part of the software (i.e., kernel and user code), without requiring dynamic linking on sensor nodes.
(4) We present the design, implementation, and demonstration of a fully functional multihop reprogramming system. Most previous work has concentrated on some of the stages of the incremental reprogramming system, but has not delivered a functional complete system.

The rest of the article is structured as follows. Section 2 surveys the related work. Section 3 gives a brief overview of various stages of Zephyr. Section 4 discusses the byte-level comparison and explains why such comparison alone is not sufficient. Section 5 presents the application-level modifications. Section 6 explains additional high-level optimizations to further reduce the delta size. Section 7 discusses the delta distribution method. Section 8 explains the testbed and the simulation setups and results. Section 9 presents the mathematical analysis of Zephyr. Section 10 concludes.

2. RELATED WORK

The problem of reconfiguration of sensor networks has been an important theme with the community. We discern three streams of work in this area. 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 Mate [Levis and Culler 2002], VM* [Koshy and Pandey 2005b], ASVM [Levis et al. 2005], and Darjeeling [Brouwers et al. 2009] 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 therefore complementary to this class.

TinyOS is the primary example of an operating system that does not support loadable program modules in the standard release. Several protocols provide reprogramming with full binaries, such as Deluge [Hui and Culler 2004], Stream [Panta et al. 2007], Freshet [Krasniewski et al. 2008], MOAP [Stathopoulos et al. 2003], and MNP [Kulkarni and Wang 2005]. To support incremental reprogramming, Jeong and Culler [2004] use Rsync to compute the difference between the old and new program images. However, because it is built on top of XNP [Inc 2003], it can only reprogram a single-hop network and does not use any application-level modifications to handle changes in function locations. We compare the delta size generated by their approach and use it with an existing multihop reprogramming protocol to compare their reprogramming time and energy with Zephyr. In Reijers and Langendoen [2003], 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 many address changes resulting in a large delta. Koshy and Pandey [2005a] use slop (empty) regions after each function in the application binary to allow functions to grow without changing the positions of other functions. However, the slop regions lead to fragmentation and inefficient use of the Flash memory. Also when the functions expand beyond the allocated space, they need to be relocated elsewhere, causing the function references to change and size of the delta script to increase. The designers of Flexcup [Marron et al. 2006] 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 each changed component, along with the new symbol and relocation tables, to the nodes. Flexcup 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. Recently, a threading architecture, called TOSThreads [Klues et al. 2009], has been incorporated in the latest release of TinyOS. In addition to providing multithreaded programming environment for sensor devices, it also allows dynamic linking of TinyOS modules. Zephyr can still be useful in such systems to transfer the difference between the old and new versions of the loadable modules.

Reconfigurability is simplified in OSes like SOS [Han et al. 2005] and Contiki [Dunkels et al. 2004]. 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 remain 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 4KB 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 [Koshy and Pandey 2005a]. Zephyr, while currently implemented in TinyOS, can also support incremental reprogramming in these OSes by enabling incremental updates to changed user and kernel modules.

Distinct from Zephyr, in Panta and Bagchi [2009], we show that further orthogonal optimizations are possible to reduce the delta size, for example, 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 Panta and Bagchi [2009], 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.

There is a long history of dynamic linking in server and desktop applications [Levine 2000] and with Java [Czajkowski 2000; Serrano et al. 2000]. While reducing the volume of code is a concern, JavaSpec and QS [Serrano et al. 2000] 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 [Levine 2000].

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 shifts in the function locations (hereafter called function shifts) so that the similarity between the two versions of the software is increased. Next the two executables are compared at byte level using a novel algorithm derived from the Rsync algorithm [Tridgell 1999]. This produces the delta script which describes the differences between the old and new versions of the software. These computations are performed on the host computer. The delta script is then transmitted wirelessly to all nodes in the network in 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 multihop manner if required. The nodes save the delta script in their external flash memory. After the sensor nodes download

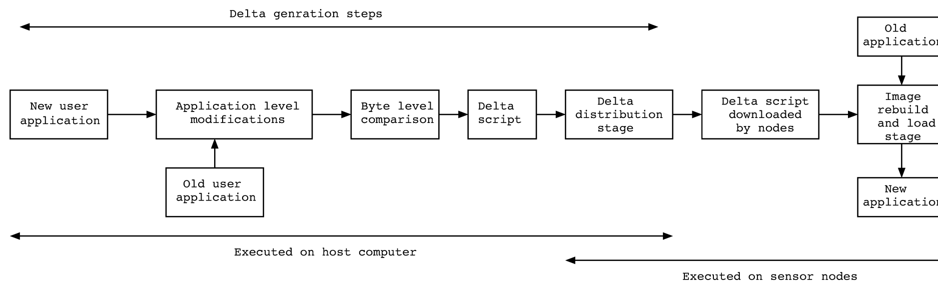


Fig. 1. Overview of Zephyr.

the delta script, they build 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 [Tridgell 1999] and then our extensions to reduce the size of the delta script that needs to be disseminated.

4.1. Application of Rsync Algorithm

The Rsync algorithm was 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 and both the sender and the 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 an expensive MD4 computation for each block of the binary image. 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 it in a hash table whose key is the checksum. (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 using a hash-table lookup. If a matching block is found, Rsync checks if their MD4 hashes also match. If MD4 hashes also match, then this block is considered as a matching block. Note that if two blocks do not have the same checksum, then MD4 is not computed for this block. This ensures that the expensive MD4 computation is done only when the inexpensive checksum matches between the two blocks. If no matching block is found then the algorithm moves to the next byte in the new image and the same process is repeated until a matching block is found. While the probability of collision is not negligible for two blocks having the same checksum, with MD4 the collision probability *is* negligible. To ensure the correctness of our scheme in the rare case when two different blocks have the same MD4 hash, Zephyr performs a byte-by-byte comparison when MD4 hashes match. Since this algorithm runs on a powerful host computer, this is not a problem.

After running this algorithm, Zephyr generates a list of COPY and INSERT commands for matching blocks and nonmatching portions respectively (the size of the nonmatching portions may not be equal to the block size):

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

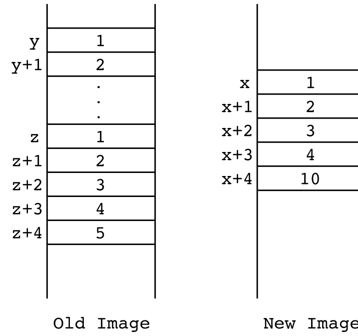


Fig. 2. Finding superblock.

The 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. The INSERT command inserts *len* number of bytes, that is, *data*, to *newOffset* of the new image. Note that this *len* is not necessarily equal to the block size or its multiple.

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 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 $\langle y \rangle$ $\langle x \rangle$ $\langle B \rangle$ and COPY $\langle y+1 \rangle$ $\langle x+1 \rangle$ $\langle B \rangle$, where B is the block size. Simply combining these 2 commands as COPY $\langle y \rangle$ $\langle x \rangle$ $\langle 2{*}B \rangle$ 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 *superblock* and the largest superblock the *maximal superblock*. The optimized Rsync algorithm finds the maximal superblock and uses that as the operand in the COPY command. Thus, optimized Rsync produces the single COPY command as COPY $\langle z \rangle$ $\langle x \rangle$ $\langle 4{*}B \rangle$ for the example just given. Figure 3 shows the pseudocode for optimized Rsync. Its complexity is $O(n^2)$, where n is the number of bytes in the image. This is not a concern because the algorithm is run on the host computer and not on the sensor nodes, and is run 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.

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 the TinyOS distribution that blinks an LED on the sensor node every second. We change the application from blinking a green LED every second to blinking it every two seconds. Thus, this is an example of a small parameter change. The delta script produced with optimized

```
/* 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.     else
9.        j=j+blockSize
10.        if (cblStretch==0)
11.          cbl=mbl
12.          cblStretch++
13.        else
14.          Empty tempCbl
15.          for each element in cbl do
16.            if (cbl.element + cblStretch == any entry in mbl )
17.              tempCbl=tempCbl U {cbl.element}
18.          if tempCbl is empty
19.            Store any one element in cbl as maximum superblock
20.            Empty cbl
21.            cblStretch=0
22.          else
23.            cbl=tempCbl
24.            cblStretch++
25. 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.*/
```

Fig. 3. Pseudocode of optimized Rsync that finds maximal superblock.

Rsync is 23 bytes long which is small and congruent with the actual amount of change made in the software.

Case 2. We added four lines of code to Blink. The delta script between the Blink application and the one with these four lines added is 2183 bytes. The 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 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 the added lines are shifted. As a result, all the calls to those functions refer to new locations. This produces several additional changes in the binary file resulting in the large delta script.

The boundaries between blocks can be defined by Rabin fingerprints as is done in Pucha et al. [2007] and Muthitacharoen et al. [2001]. 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 a Rabin fingerprint can be a substitute for a 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 these 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 calls to these functions change, and 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 (8KB 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 8KB than at 20 bytes.

5. APPLICATION-LEVEL MODIFICATIONS

The size of the delta script produced by a byte-level comparison is not always consistent with the extent of the change made in the software. This is a direct consequence of neglecting the structure of the code at the application level of the software and using only the binary comparison for generating the delta script. So we need to make modifications at application level so that the subsequent stage of byte-level comparison produces delta script that is 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 Koshy and Pandey [2005a]. 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, and thus is not desirable for memory-constrained sensor nodes. Also, this approach creates a host of complex management issues such as what should be the size of the slop region (possibly different for different functions), and what should be done with the empty memory space caused by relocation of functions when they expand beyond the assigned slop region. Choosing too large of a slop region means wasting too much memory, and too small a slop region means functions frequently need to be relocated, leading to large differences in the binary images. Another way of mitigating the effect of function shifts is by making the code position independent [Han et al. 2005]. 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 structural adjustments at 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 fun_1, fun_2, and fun_n 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 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 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 $fun_1, fun_2, \ldots, fun_n$ are replaced by jumps to *fixed locations* $loc_1, loc_2, \ldots, loc_n$, 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 loc_1 acts like an indirection table. In this table, the actual jumps to the functions are made. When the call to the actual function returns, the flow of the control is directed back to the line following the call to $loc_x (x = 1, \ldots, n)$. 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 $fun_1, fun_2, \ldots, fun_n$ 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, the linker checks if the indirection table in the old file contains the entry

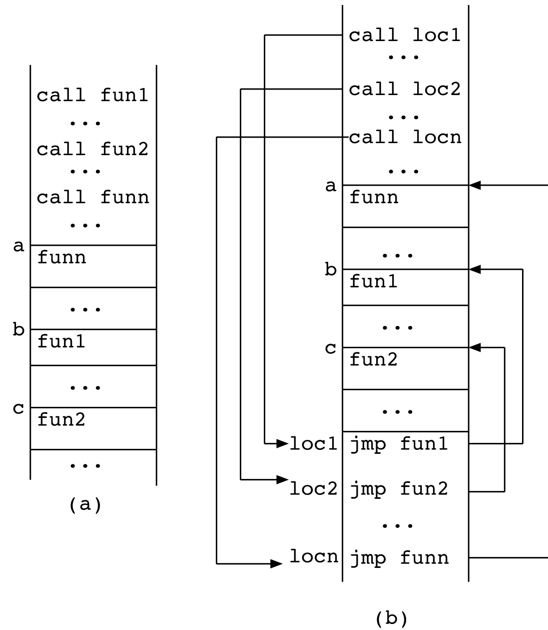


Fig. 4. Program image: (a) without indirection table and (b) with indirection table.

for that function (we also supply the old file (Figure 4(b)) as an input to the executable generation process). If it does, then it creates an entry for this function in the indirection table in the new file at the same location as in the old file. Otherwise it assigns a slot in the indirection table for the 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 these slots to the rootless functionss. If there are still some rootless functions without a slot, then the indirection table is expanded with new entries to accommodate these rootless functions. 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 the new application than in the old one.

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 the 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 does without an 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 superblock 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 preceding modifications. We use linker command language to implement function call indirections. Zephyr does introduce 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).

As mentioned before, Koshy and Pandey [2005a] allocate fixed extra space for each function. This has some drawbacks. First, it wastes program memory. Second, if a function expands beyond its allocated space, it needs to be reallocated somewhere else in the program memory. This creates "holes" in the program memory. Note that other existing functions cannot be moved to these holes since this changes the address of those functions and the delta script becomes larger. Reallocation of the function also causes the address of the reallocated function to change between the old and new versions of the software, increasing the size of the delta script. Koshy and Pandey [2005a]'s approach can be modified such that when a function (say $f1$) expands beyond the allocated region (say $R1$) in program memory, we can use a "jmp" statement at the end of $R1$ to direct the flow of control to a new region, say $R2$. This modification to Koshy and Pandey [2005a] uses Zephyr's concept of indirection, but in a less elegant and more complicated manner because of the following reasons. Let us assume that in the next round of reprogramming the function $f1$ expands further such that it no longer fits in the added region $R2$. So we again need to redirect the flow of the control from the end of region $R2$ to another region $R3$. With this approach, a single function would be scattered at many disjoint places in program memory. This is conceptually not elegant, and also increases execution time due to the possible chaining of a sequence of jumps. We believe that our approach is more elegant and scalable because it requires only one level of indirection. Furthermore, if the function $f1$, which is currently using regions $R1, R2, R3, \ldots, Rn$, shrinks such that it only needs a small part of $R1$, then this modified scheme leaves holes in the program memory. Note that "old" functions cannot be moved to these holes since this changes the locations of these functions, and hence the size of the delta script would be increased. The functions, which are newly added to the latest version of the software, can be moved to these holes. However, it adds an extra management overhead; we need to keep track of which regions in program memory are free, how large is each free region, how much program memory space is needed by newly added function, etc. Also in this modified scheme, it is unclear how much space should be allocated to added regions like $R2$. If a small region is allocated, the probability of the function expanding beyond $R2$ in the next round of reprogramming becomes high. If a large region is allocated, program memory is wasted. We believe that Zephyr provides a simple, elegant, and yet very effective solution compared to the aforesaid modification to Koshy and Pandey [2005a].

5.2. Pinning the Interrupt Service Routines

It should be noted that changes in the application software can cause changes not only in the positions of the user functions but also the positions of interrupt service routines. Such routines are not explicitly called by the user application. In most microcontrollers, there is an interrupt vector table at the beginning of the program memory, typically 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 described earlier 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. This

approach is based on the assumption that interrupt service routines generally do not change. If interrupt service routines change, it does not cause a correctness problem, but causes the delta script to be larger.

6. METACOMMANDS FOR COMMON PATTERNS OF CHANGES

After the delta script is created through the aforesaid 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

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>
COPY <oldOffset=O3> <len=L3> <newOffset=N3>
INSERT <newOffset> <len=l1> <data3>
```

and so on. Let L_i indicate a large value, and l_i a small value. Here, small INSERT commands are present in between large COPY commands. 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 $O1 + L1 + l1 = O2$, $O2 + L2 + l1 = O3$, and so on. In many software change cases that we evaluated, we found that two blocks in two versions of the image match perfectly, except at few places where a single byte operand of some instructions differs. In other words, if the blocks corresponding to INSERT commands with small *len* had matched, we would have obtained a very large superblock. So Zephyr replaces 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* $(i = 1,2\ldots, n)$, *numInserts*=*n* is the number of *(addr,data)* pairs, and *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 $len = L1 + l1 + \cdots + Ln$ 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* $(i = 1, 2,\ldots, n)$.

6.2. REPEAT Command

This command is useful for reducing the number of bytes in the delta script that are needed to transfer the indirection table. As shown in Figure 4(b), the indirection table consists of the pattern *jmp fun1, jmp fun2, ...* where the same string of bytes (say *S1* = *jmp*) repeats, with only the addresses for *fun1, fun2*, etc., changing between them. So Zephyr uses the following command to transfer the indirection table.

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

This command puts the string *S1* at offset *newOffset* in the new image followed by *addr1*, then *S1*, then *addr2*, and so on until *addrn*. Note that the CWI command could have also been used for this case, but since string *S1* is fixed, fewer bytes are needed 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, the COPY command is used to transfer 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 Zephyr does not need to specify the offset in the new file in any of the previous commands. Monotonic means Zephyr always writes at location x of the new image before writing at location y, for all $x < y$. Instead of the new offset being provided, a counter is maintained and incremented as the new image is built, and the next write always happens at this counter, allowing the *newOffset* field to be dropped from all the commands.

We find that for Case 2, where some functions are shifted due to the 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 is used without application-level modifications. The size of the delta script without the metacommands 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 this case there is 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 a small delta script even in such extreme cases, it is necessary that the binary image itself be small. Since the size of the binary image transmitted by Stream [Panta et al. 2007] is almost half the size of that of Deluge [Hui and Culler 2004], 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 a code update is done. The reason for this requirement in Deluge is that the reprogramming component needs to be running on the sensor nodes all the time so that the nodes can be receptive to future code updates and these nodes are not capable of multitasking (running more than one application at a time). Stream solves this problem by storing the reprogramming component in the external flash and running it on demand, whenever reprogramming is to be done.

Distinct from Stream, 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

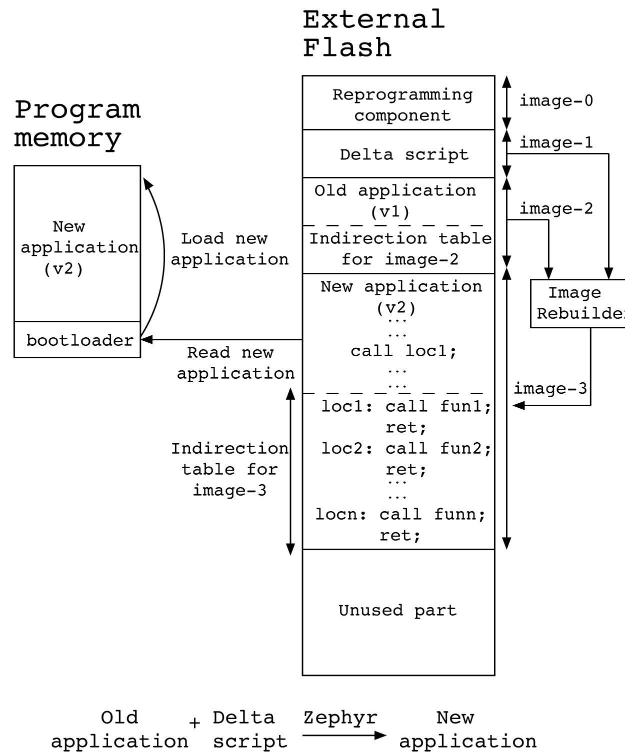


Fig. 5. Image rebuild and load stage. The right side shows the structure of external flash in Zephyr.

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

As mentioned before, Zephyr uses a simple flooding scheme to send reboot commands to all nodes in the network. This method is generally sufficient to ensure that all nodes receive the reboot command. Note that every node broadcasts the reboot command once, before rebooting from the reprogramming component (i.e., img-0 as shown in Figure 5). So, even if a node does not receive the reboot command from one of its neighbors due to, say, poor link quality between them, it may receive the command from other neighbor(s). However, due to very poor link reliability with all neighbors, or hardware/software faults, some nodes may not be able to receive the reboot command. To address this issue, Zephyr provides an “eventual consistency” property, similar to our previous work, Stream [Panta et al. 2007].

Let us call the nodes which do not receive reboot command from any of its neighbors the *faulty nodes*. In Zephyr, sensor nodes use the Trickle [Levis et al. 2004] algorithm to periodically advertise their metadata consisting of the version number of the program images that they possess (img-0 through img-3 in Figure 5). When the faulty nodes eventually recover from their faults, they will receive the advertisement message from their neighbors and learn that they do not possess the latest version of the binary image.

This causes the faulty node to broadcast its advertisement message immediately and reboot from the reprogramming component (img-0). If the neighbors of the faulty node are executing img-0 (this happens if they have not yet completed downloading the new version of the application image), the faulty node starts downloading the new image from its neighbors according to the Deluge algorithm. If the neighbors of the faulty node are not executing img-0 (this happens if they have already downloaded the new version of the application image and have already started running the new image), they find that their faulty neighbor is not up-to-date and hence reboot from img-0 to bring the faulty node up-to-date. Then the faulty node downloads the delta and and builds the new image. Note that due to inherent failure-prone nature of sensor networks, many prior works [Levis et al. 2004; Hui and Culler 2004; Panta et al. 2007] have shown that eventual consistency is a satisfactory solution. Hence Zephyr follows this approach.

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 the code update has completed 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 begin executing 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 does not, however, pose a correctness issue. After n_1 reboots from the new code, it will switch again to the reprogramming state when it receives an advertisement from n_2. This, however, 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 built image will be stored as image 2. As we will show in Section 8.3, the time to build 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 assumption behind any incremental reprogramming protocol is that in practice, the software changes are

generally small and 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 in 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 the 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, that is, the last packet of the last page may be padded with zeros. But this results in sufficiently small wastage that we did not feel justified in introducing the additional complexity of dynamic packet sizes. Our scheme can be further modified to advertise the actual number of packets in the last page to minimize the wastage. For example, in the case where the delta script has 1105 bytes, it would transfer two 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. Change the Blink application from blinking a green LED every second to blinking it every 2 seconds.

Case 2. Add a few lines added to the Blink application.

Case 3. Change the 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. Change CntToLeds to CntToLedsAndRfm: CntToLeds is the same as CntToLedsAndRfm except that the counting sequence is not transmitted over radio.

Case 5. Change Blink to CntToLeds.

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

Case 7. Change 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 [eStadium] which is 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, and so forth. We considered a subset of the changes that the software had actually gone through during various stages of refinement of the application.

Case A. Change an application that samples battery voltage and temperature from an MTS310 [xbow] sensor board to one where a few functions are added to also sample the photo sensor.

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 preceding 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 the 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, where the software has changed completely (the goal of the software has changed). For example, in Case 6, the goal of the software is changed from periodically blinking an LED to perform routing.

These classes of software changes are based on the degree of the change that the software has undergone at application level. Many of the previous cases involve changes in the OS kernel as well. Strictly speaking, in TinyOS, there is no separation between OS kernel and application. The two are compiled as one large 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 temperature and battery voltage to 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 [jarsync], a Java implementation of the Rsync algorithm, to achieve the optimizations mentioned in Section 4.2. From here onward, by "semioptimized Rsync," we mean the scheme that combines two or more contiguous matching blocks into one superblock. It does not necessarily produce the maximal superblock. By "optimized Rsync" we mean our scheme that produces the maximal superblock but without the application-level modifications.

As shown in Figure 6, the size of the delta script produced by either Rsync or optimized Rsync depends on the block size used in the algorithm. Recollect that the comparison is done at the granularity of a block. As expected, Figure 6 shows that the size of the delta script is largest for Rsync and smallest for optimized Rsync. Figure 6 also shows that as the block size increases, the size of the delta script produced by Rsync and semi-optimized Rsync decreases until a certain point after which it has an increasing trend. The size of the delta script depends on two factors: (1) the number of

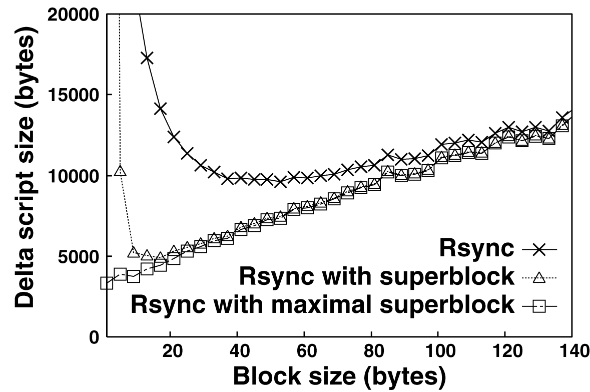


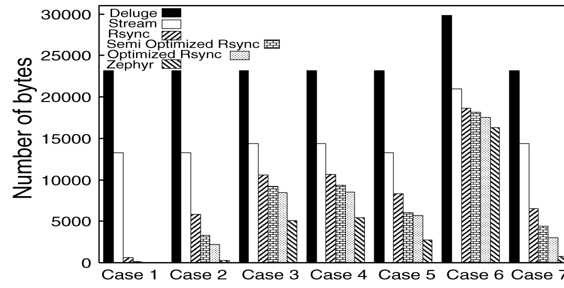
Fig. 6. Delta script size versus block size.

commands in the delta script and (2) the size of the 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 superblock. 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 size of new data to be inserted. As a result, the delta script size increases.

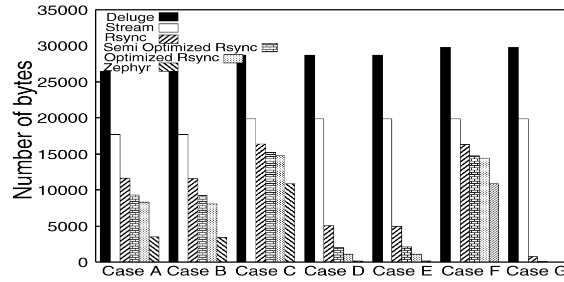
For optimized Rsync, there is a monotonic increasing trend for the delta script size as block size increases. There are, however, some small oscillations in the curve, as a result of which the optimal block size is not always one byte. The small oscillations are because increasing the block size decreases the size of maximal superblocks and increases the size of data in INSERT commands. But sometimes the small increase in the size of the 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.

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 sensor nodes. A small delta script translates to less reprogramming time and energy due to fewer packet transmissions over the network and small number of external flash writes on the node. Figure 7 and Table I compare the delta script produced by Deluge, Stream, Rsync, semi-optimized Rsync, optimized Rsync, and Zephyr. Table I also compares Zephyr with *ZephyrWOMetaCmds*: the case where all application-level modifications are used except metacommands. 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 significantly reducing the size of delta script compared to optimized Rsync. Optimized Rsync takes up to nine



(a) TinyOS software change cases



(b) eStadium software change cases

Fig. 7. Size of data transmitted for reprogramming.

Table I. Comparison of Delta Script Size of Various Approaches

	Deluge : Zephyr	Stream : Zephyr	Rsync : Zephyr	SemiOptRsync : Zephyr	OptRsync : Zephyr	ZephyrWOMetaCmds : Zephyr
Case 1	1400.82	779.29	35.88	6.47	1.35	1.35
Case 2	85.05	47.31	20.81	11.75	7.79	1.99
Case 3	4.52	2.80	2.06	1.80	1.64	1.38
Case 4	4.29	2.65	1.96	1.72	1.57	1.30
Case 5	8.47	4.84	3.03	2.22	2.08	1.39
Case 6	1.83	1.28	1.14	1.11	1.07	1.05
Case 7	29.76	18.42	8.34	5.61	3.87	1.52
Case A	7.60	5.06	3.35	2.66	2.37	1.6
Case B	7.76	5.17	3.38	2.71	2.37	1.61
Case C	2.63	1.82	1.50	1.39	1.35	1.16
Case D	203.57	140.93	36.03	14.36	7.84	2.33
Case E	243.25	168.40	42.03	17.66	9.01	2.43
Case F	2.75	1.83	1.50	1.36	1.33	1.18
Case G	1987.2	1324.8	49.6	6.06	1.4	1.4

Deluge, Stream and Rsync represent prior work.

times more bytes than Zephyr. These cases correspond to function shifts in the software. As a result, application-level modifications have great effect in these 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, a significant effect in the very large software change case (in terms of absolute delta size reduction), and a small effect on the very small software change cases.

Comparison with other incremental approaches. Rsync represents the algorithm used by Jeong and Culler [2004] to generate the delta by comparing the two executables

without any application-level modifications. We find that Jeong and Culler [2004] produce up to 49 times larger delta script than Zephyr. As mentioned before, in Koshy and Pandey [2005a]'s approach, the program memory is fragmented and used less efficiently than in Zephyr. Flexcup [Marron et al. 2006], though capable of incremental linking and loading on TinyOS, generates high traffic through the network because of large symbol and relocation tables. Also, Flexcup is implemented only on an emulator whereas Zephyr runs on the real sensor node hardware.

To describe function shifts, Reijers and Langendoen [2003] augment the delta script with a *patch list command*: a list of {*begin address, end address, offset*}-tuples, each of which describes the offset by which each function is shifted in the new version of the user application. More specifically, this command says that all functions which lie between {*begin address, end address*} are shifted by the *offset* bytes in the new version. The patch list command incurs an overhead of $2 + 6 * count$ bytes (2 bytes for *count*: the number of patch lists; 2 bytes each for *begin address, end address, and offset*). The patch list command is useful for small software change scenarios where many functions are shifted by the same offset and thus can be described by a few patch lists. However, for many practical software change cases, different functions are shifted by different offsets (e.g., software change cases 3, 4, 5, 6, 7, B, C, and F in our experiments). In such cases, the overhead due to the patch list command can be significant because of the increase in the number of patch lists required to express the shifts in all functions between old and new versions of the code.

The patch list command can also significantly increase the size of the delta script (although patch list command can be thought of as a part of the delta script, we categorize COPY, INSERT commands as delta script, and patch lists as patch list command, for convenience of explanation). Note that in Reijers and Langendoen [2003], when executing a COPY command, a node checks for each word that it copies if the previous word is the opcode of a patchable instruction, the instruction that uses function address. If so, and if the copied word lies in the range covered by one of the entries in the patch list, the node adds the corresponding offset to the copied word. However, implementing this is difficult as well as architecture dependent. This is because it is not always possible to distinguish a patchable instruction by just looking at the opcode in the binary executable. For example, while pushing a task in the task queue, TinyOS uses *ldi* statements to load registers *r24* and *r25* with the address of the task (a function). By just looking at the binary code, it is not possible to say whether the operand of *ldi* instruction is a function address or say some constant. As a result, implementation in Reijers and Langendoen [2003] does not look for patchable instruction. Instead it patches all binary words that lie in the {*begin address, end address*} range. As a consequence, the node ends up performing many "mispatches" as acknowledged by the authors. To correct these mispatches, Reijers and Langendoen [2003] use REPAIR commands (these are used for other purposes also) in the delta script, thereby increasing the size of the delta script. This is illustrated by the fact that for most of the software change cases used in Reijers and Langendoen [2003], the decrease in the size of the delta script is not as significant as in our experiments. Furthermore, Reijers and Langendoen [2003] assume that the function address is always preceded by an instruction opcode (for simplifying the identification of the patchable instruction). This, however, is dependent on the microcontroller architecture and may not be true for many architectures. For example, in ATmega128, as mentioned earlier, the second operand of the *ldi* statement can be the function address. We believe that Zephyr offers an elegant solution without all these complications and overhead.

Apart from these core differences, Reijers and Langendoen [2003] do not present an implementation of a complete incremental reprogramming system. They focus mainly on the generation of the difference, and not on actual dissemination. Zephyr, on the

other hand, is a complete usable incremental reprogramming system. Furthermore, Reijers and Langendoen [2003] present an evaluation of their system with very few software change scenarios, whereas in this article, we present a comprehensive evaluation of Zephyr.

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. Furthermore, 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, Zephyr's advantage will be even greater since there will be more function calls. Zephyr's advantage will be minimal if the software change does not shift any function. For such a change, the advantage will be only from the optimized Rsync algorithm. But such software changes are very rare, for example, 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 all following functions to be shifted.

8.3. Testbed Experiments

We perform testbed experiments using Mica2 [xbow] 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 adjacent nodes. Due to fluctuations in transmission range, occasionally a nonadjacent node will receive a packet. In our experiments, if a node receives a packet from a nonadjacent node, it is dropped, to achieve a truly multihop network. This kind of software topology control has been used in other works also [Kamra et al. 2006; Panta et al. 2008]. A node situated at one corner of the grid or end of the line acts as the base node. We provide a quantitative comparison of Zephyr with Deluge [Hui and Culler 2004], Stream [Panta et al. 2007], Rsync [Jeong and Culler 2004], and optimized Rsync without application-level modifications. Note that Jeong and Culler [2004] reprogram only nodes within one hop of the base node, but we used their approach on top of a multihop reprogramming protocol to provide a fair comparison. The metrics for comparison are reprogramming time and energy. We perform these experiments for grids of size 2×2 to 4×4 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 [2004].

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 build the new image. The 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 that 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 Panta et al. [2008], which achieves this with minimal overhead traffic.

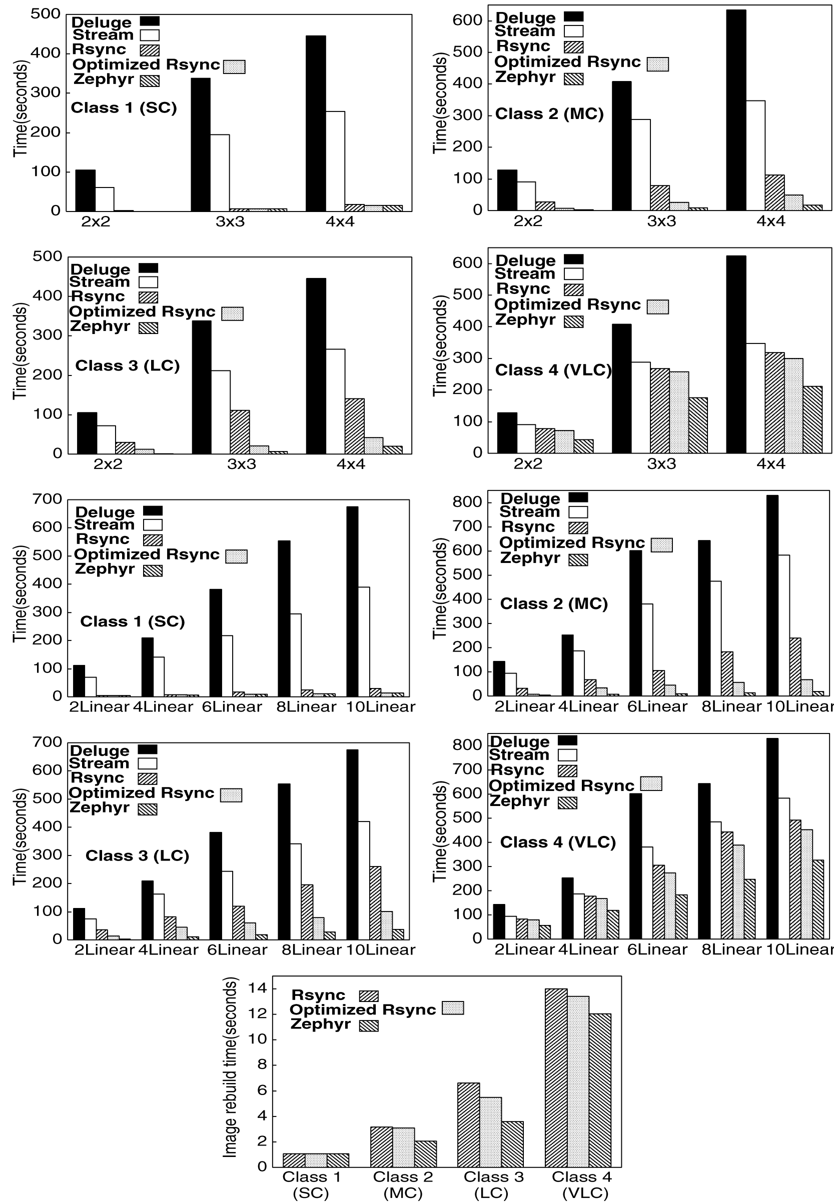


Fig. 8. Comparison of reprogramming times for grid and linear networks. The last graph shows the time to rebuild the image on the sensor node.

Figure 8 (except for the last graph) compares reprogramming times of other approaches with Zephyr for different grid and linear networks. Table II 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 nonincremental reprogramming protocols like Deluge and Stream significantly for all the cases. Zephyr is also up to 12.78 times faster than Rsync, the approach of Jeong and Culler [2004]. This illustrates that the Rsync optimization and

Table II. 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

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 changes cause function shifts. So, these results show that application-level modifications greatly mitigate the effect of function shifts and reduce the reprogramming time significantly. For the 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 little room for improvement. 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.

8.3.2. Reprogramming Energy. The most important factors that contribute to energy consumption during reprogramming are radio communications and flash writes. Obviously, the energy cost due to radio transmissions and receptions are directly proportional to the number of packets that are transmitted by all nodes in the network for reprogramming. As mentioned earlier, the downloaded delta script is first stored in the external flash by each sensor node. The new image, which is built using the delta script and the old image, is also stored in the external flash. Then the new image is loaded from from external flash to the flash program memory by the bootloader. Thus the number of flash program memory accesses for loading the new image from external flash is independent of the number of packets received by the sensor node, and is same for all reprogramming protocols because ultimately each protocol is creating the same new version of the program. However, the number of external flash accesses (to store the delta script) is directly proportional to the number of packets received by the sensor node. Thus, the total number of packets transmitted by all nodes in the network during reprogramming is a good measure of the energy cost due to radio transmissions and receptions, as well as flash writes. In this section, we first compare different protocols in terms of total number of packets transmitted by all nodes in the network for reprogramming.

Figure 9 and Table III 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 Jeong and Culler [2004]. 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

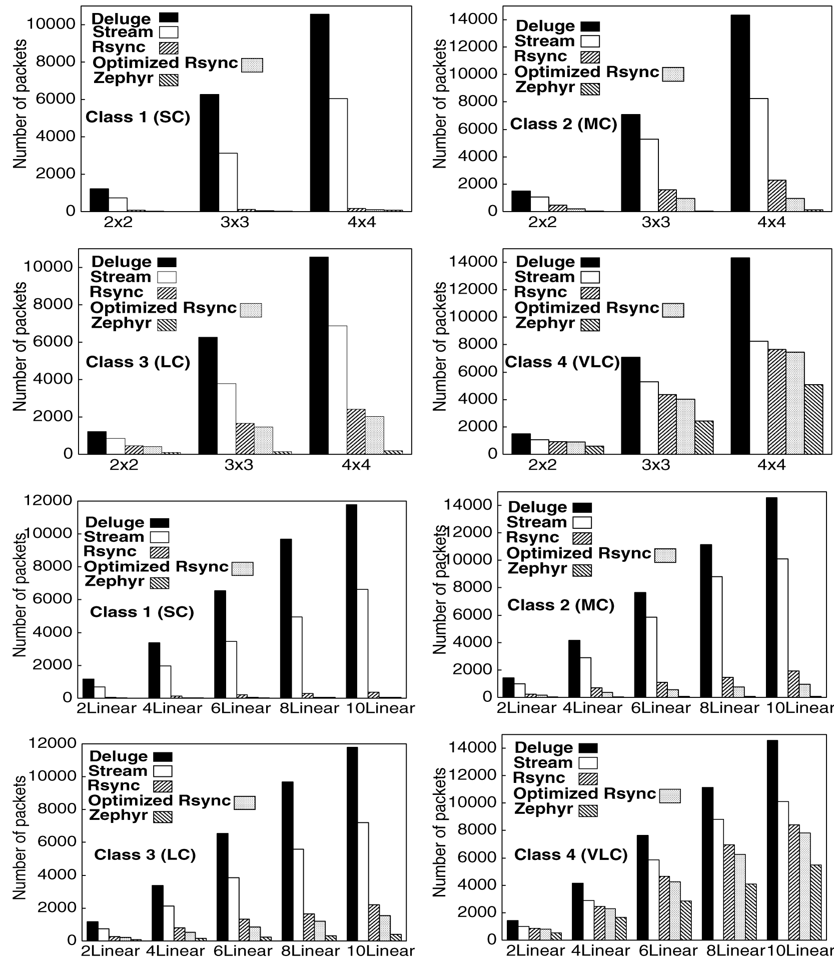


Fig. 9. Comparison of number of packets transmitted during reprogramming.

Table III. 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

the software change without shifting any function, the application-level modifications achieve a 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.

As mentioned before, radio communication is a major source of energy consumption during reprogramming. The energy cost due to radio communication can be grouped into three categories: (1) Idle listening energy cost, E_1: It is the energy consumption due to nodes listening to the wireless medium when there is no packet transmission in their neighborhood. (2) Energy cost due to transmission and reception of *unnecessary*

packets, E_2: By unnecessary packets, we mean (a) the corrupt packets, (b) code packets that a node has already downloaded, and (c) the code packets which a node does not store upon reception. Note that in Zephyr, like in Deluge, a node downloads code packets in a *monotonic order*; it downloads packets of a page x before packets of page y, for all $x < y$. This is done to avoid the state maintenance overhead. (3) Energy cost due to transmission and reception of *necessary* packets, E_3: By necessary packets, we mean the code packets that a node stores upon reception.

The underlying MAC protocol affects energy costs E_1, E_2, and E_3. Low-Power Listening (LPL) MAC protocols [Buettner et al. 2006; Polastre et al. 2004; El-Hoiydi and Decotignie 2005] cause the sensor nodes to put the radio transceiver to sleep mode for most of the time and wake it up periodically for a short period of time to sample the channel to check if there is any radio transmission in their neighborhood. Idle listening energy cost is significantly reduced by duty-cycling LPL MACs because the time duration for which the radio is turned on without any packet transmission or reception is limited to short channel sampling periods. The asynchronous LPL MACs (like BMAC [Polastre et al. 2004] and XMAC [Buettner et al. 2006]) avoid the need for time synchronization among nodes by having the transmitter transmit a (long) preamble sequence before each packet transmission such that the preamble duration is at least as long as the sleep period of the nodes. The long preamble ensures that when a node wakes up to sample the channel, it is guaranteed to to hear the preamble. When a node hears a preamble, it turns its radio on until the packet is received. Thus the energy cost is incurred not only during packet transmission and reception, but also during preamble transmission and reception.

The energy costs E_2 and E_3 are due to transmission and reception of both preambles and actual packets (unnecessary packets for E_2 and necessary packets for E_3). Note that E_2 also includes the energy cost due to a node keeping its radio on after receiving a preamble for a packet that it later finds it is not interested in. Let N_P be the total number of packets transmitted by all nodes in the network during reprogramming. It is the sum of all necessary and unnecessary packets. Clearly, the energy costs E_2 and E_3 are directly proportional to N_P. Thus the total number of packets transmitted by all nodes in the network during reprogramming provides a measure of E_2 and E_3. From Figure 9, we see that Zephyr reduces the number of packets transmitted compared to other protocols.

Each node spends some time (say t) in transmitting and receiving necessary and unnecessary packets. t also includes preamble duration. During rest of the time period (say, $T_R - t$, where T_R is the total reprogramming period), the node samples the channel periodically (without detecting radio transmission in its neighborhood) according to its duty-cycling schedule. Thus, the reprogramming period T_R provides a measure of idle energy cost E_1. Figure 8 shows that Zephyr requires significantly less time to reprogram the network compared to other protocols.

Next we present a simplified mathematical analysis to complement the qualitative argument that we presented earlier: the total number of packets transmitted by all nodes in the network is a measure of the energy costs E_2 and E_3, and the reprogramming period is a measure of the energy cost E_1. To simplify the analysis, we consider a single-hop network. Let us suppose this network takes T_R time to be reprogrammed. Suppose that N_P is the total number of packets transmitted by all nodes in the network during reprogramming. The upper bound[1] of the time period t spent by each node in the network for receiving and transmitting necessary and unnecessary packets (including

[1]This is an upper bound because (a) a node may not turn its radio on for the entire preamble duration if it detects the preamble at an instant other than the exact start of the preamble, and (b) during the periodic channel sampling, a node may miss the preamble because of bad link conditions.

preamble) is

$$t = N_P * t_p, \tag{1}$$

where t_p is the time to transmit a single packet. t_p is given by

$$t_p = t_{PR} + t_{DATA}, \tag{2}$$

where t_{PR} is the time to transmit the preamble and t_{DATA} is the time to transmit the actual (data) packet. To calculate t_{PR} and t_{DATA}, let us consider XMAC, which is the LPL MAC used by TinyOS. As mentioned before, in LPL MAC schemes, before transmitting a packet, each node transmits a preamble which is at least as long as the sleep period to ensure that its neighbor receives the preamble and hence keeps its radio on to receive the actual data packet. In XMAC, the length of the preamble is reduced (for unicast packets) because during the preamble period, a sender node transmits a series of strobe packets containing the receiver's address with a gap between the successive strobe packets. When the receiver node wakes up and detects the strobe packet, it sends an ACK during the gap between two successive strobe packets. Upon receiving the ACK, the sender node stops transmitting the strobe packets and immediately sends the actual data packet. Note that during reprogramming, most of the packets are broadcast (except request packets). Hence, in XMAC, the preamble strobe packets have to be sent for the entire sleep period to make sure that all neighbors receive the broadcast packet. As a result, the preamble duration (t_{PR}) in XMAC is equal to the sleep period. The sleep period depends upon duty cycle and time required by the radio transceiver to sample the channel. Let t_{cs} be the channel sampling period. Since duty cycle is given by $d_c = t_{cs}/(t_{cs} + t_{PR})$, thus the preamble period is given by

$$t_{PR} = \frac{t_{cs}(1 - d_c)}{d_c}. \tag{3}$$

t_{cs} should be long enough to ensure that it does not lie in the gap between the preamble strobe packets, and thus the receiver does not miss the preamble strobe packet. For this, t_{cs} should be slightly longer than the gap between the strobe packets. Let us approximate t_{cs} as

$$t_{cs} = T_{ACK} + T_{TA}, \tag{4}$$

where T_{ACK} is the time required by the receiver to transmit the ACK packet and T_{TA} is the turnaround time: time required by the radio transceiver to switch from receive mode to transmit mode. Time to transmit actual data packet depends on the size of the packet and the data rate supported by the radio transceiver. Assuming packet size = 36 bytes (the default packet size in TinyOS) and data rate = 250kbps (for IEEE 802.15.4-based radios like CC2420), time to transmit a single data packet is 1.152 ms. Thus total time to transmit a single packet, t_p, is the sum of the preamble and actual data packet durations.

$$t_p = 1.152 * 10^{-3} + \frac{(T_{ACK} + T_{TA})(1 - d_c)}{d_c} \tag{5}$$

Substituting t_p from Eq. (5) in Eq. (2), we get t, the time spent by each node for receiving and/or transmitting necessary and unnecessary packets. Thus the energy costs E_2 and E_3 incurred by all nodes in the network for transmitting and receiving necessary and unnecessary packets is

$$E_2 + E_3 = t * P * N, \tag{6}$$

where P is the transmission or receive power and N is the total number of nodes in the network. Note that for most of the currently used radio transceivers like CC2420

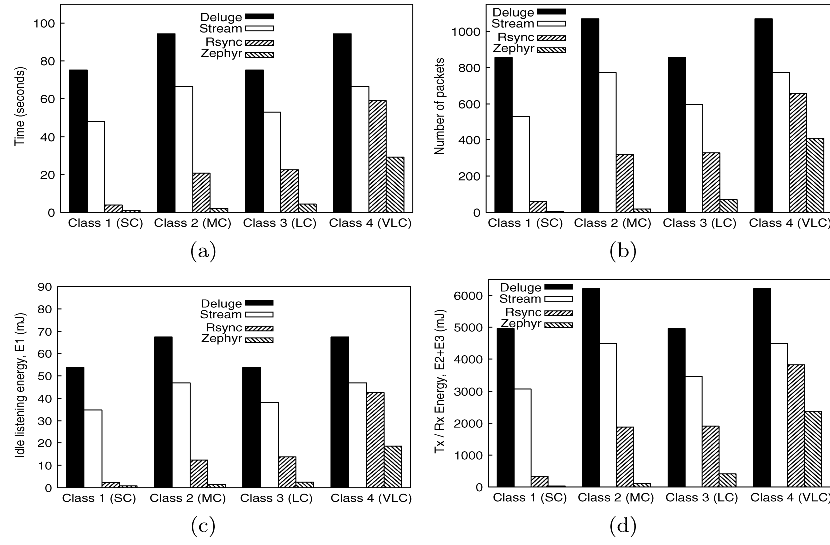


Fig. 10. Comparison of Zephyr with other approaches for a 5 node single-hop network. (a) reprogramming time; (b) number of packets transmitted during reprogramming; (c) idle energy (E_1); (d) receive/transmit energy ($E_2 + E_3$). MAC duty cycle is 2%.

Table IV. Parameter Values Used for Analysis, based on CC2420 Datasheet

Parameter	Value
Data rate	250 Kbps (for IEEE 802.15.4 radio)
T_{TA} (Turnaround time)	192 us
T_{ACK} (Time to send an ACK)	352 us (Time to transmit 11 byte ACK, based on IEEE 802.15.4 standard
P (Transmit or Receive power)	52.2 mW

[cc2420], the transmit and receive powers are almost equal. For example, for CC2420, transmit power is 52.2 mW (at 0 dBm) and receive power is 56.4 mW. Hence, we have simplified the calculation and used a single value for transmit and receive power.

$T_R - t$ is the time period spent by nodes in not receiving or sending packets. During this time it incurs idle listening energy cost due to periodic sampling of the wireless medium. The idle listening energy cost E_1 for N nodes is given by

$$E_1 = (T_R - t) * P * d_c. \tag{7}$$

We conduct testbed experiments to find reprogramming time and total number of packets transmitted by all nodes in the network during reprogramming for a 5-node single-hop network of mica2 nodes. Figure 10(a) and (b) show reprogramming time and number of packets transmitted, respectively, for 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). We see that Zephyr significantly reduces reprogramming time and number of packet transmissions. We then use the mathematical analysis presented previously to find the transmission and reception energy cost ($E_2 + E_3$) and idle listening energy cost (E_1). Parameter values used for our computations are shown in Table IV. All of these values are computed using CC2420 datasheet [cc2420]. We use 2% duty-cycle value in our calculations. The results are plotted in Figure 10(c) and (d). As expected, they show that transmission/reception and idle listening energy costs are directly proportional to the total number of packets transmitted by all nodes in the network during reprogramming and reprogramming time, respectively.

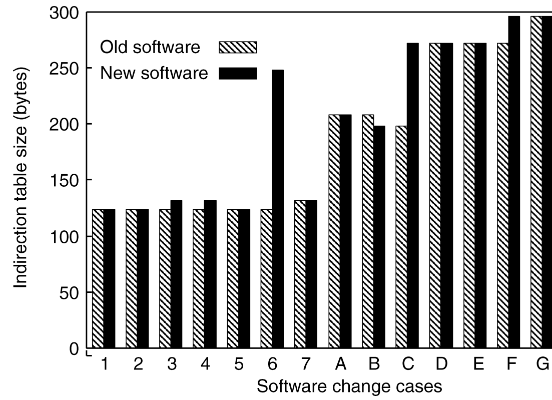


Fig. 11. Size of indirection table for various software change cases.

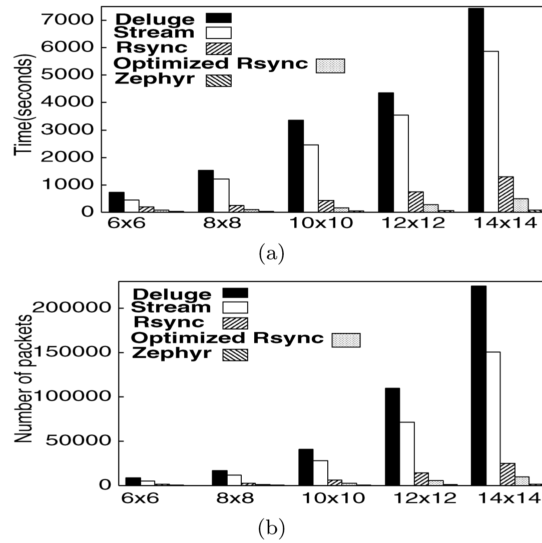


Fig. 12. Simulation results for (a) reprogramming time and (b) number of packets transmitted during reprogramming (Case D, i.e., Class 2 (MC)).

8.4. Size of Indirection Table

Zephyr needs to allocate extra space in program memory for storing the indirection table. Figure 11 shows the size of the indirection table for various software change cases mentioned before. The size of the indirection table is directly proportional to the number of functions in the software.

8.5. Simulation Results

We perform TOSSIM [Levis et al. 2003] simulations on grid networks of varying size (up to 14×14) to demonstrate the scalability of Zephyr and to compare it with other schemes. Figure 12 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 [Jeong and Culler 2004], and optimized Rsync without application-level modifications, respectively. Also, Deluge, Stream, Rsync [Jeong and Culler 2004], and optimized Rsync transmit up to 146.4, 97.9,

16.2, and 6.4 times more packets than Zephyr, respectively. Most software changes in practice are likely to belong to this class (moderate change), and 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.

8.6. Best- and Worst-Case Scenarios

In the best case, when there is no change in the software, Zephyr needs a single COPY command in the delta script as follows.

```
COPY 0 <image_size>
```

Here 0 is the offset in the old image. It says "copy *image_size* number of bytes from old offset 0 to the new image". Note that we do not need to mention the offset in the new image as mentioned in Section 6.3. This delta script takes 5 bytes. However, this best case example is just hypothetical because when there is no change in the software, there is no need for (incremental) software update. So, we do not include this hypothetical case scenario in the experimental evaluation section. The small change (SC), moderate change (MC), large change (LC), and very large change (VLC) software change cases evaluated in this article represent the realistic spectrum of the best-case to the worst-case scenarios, with SC representing the best case and VLC representing the worst case. Furthermore, we believe that most of the software change cases are of SC or MC types and thus they represent the average cases. Ideally, the worst-case scenario would be the one where the two versions of the software are completely different. However, in practice, some portions of the software (e.g., driver code, core operating system code, etc.) rarely change, or even if they change, they are very rarely entirely different from the previous version. So, the VLC cases presented in this article are good measures of practical worst-case scenarios for Zephyr. In the hypothetical worst-case example where the two versions of the software are completely different, Zephyr uses the following command in the delta script.

```
INSERT <software_size> <entire_software_image>
```

In this hypothetical example, the size of the delta script in Zephyr is 3 bytes more than the size of the binary image. The entire binary image is what would be sent by earlier works, such as Stream [Panta et al. 2007].

9. ANALYSIS

The main idea of function call indirections is that if the positions of N_S functions have changed in the new software and these shifted functions are called C_S times in the new program code, Zephyr's technique of function call indirections causes only N_S function calls in the indirection table to be different between the two versions of the software. On the other hand, C_S function call statements are different in the baseline case. Typically $C_S \gg N_S$ and thus function call indirection reduces the size of the delta script significantly. The effectiveness of function call indirection depends on the number of times the shifted functions are called in the new program.

In this section, we analyze the effect of function call indirection and the metacommands in reducing the size of the delta script. First we define a few terms. Two function call statements in the old and new programs are said to be *identical* if they call the same function. The target addresses in the identical function call statements may or may not be same. We use an attribute called *Preserved Similarity Index (PSI)* to quantify the amount of similarity preserved in the new version of the software with respect

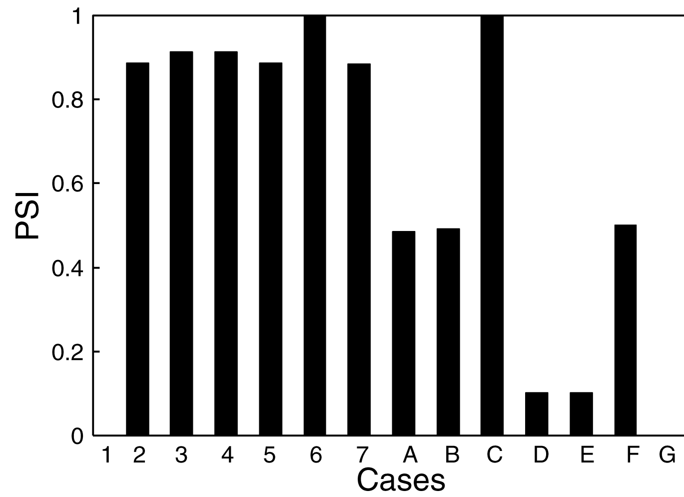


Fig. 13. Preserved Similarity Index (PSI) for different software change cases.

to the old version due to function call indirections. We define PSI as

$$PSI = \frac{C_S}{C}, \tag{8}$$

where C_S is the number of identical function call statements in the new software that would have different target addresses than those in the old software (due to the change in the position of the corresponding functions) if function call indirections were not used. C is the total number of identical function call statements in the old and the new images. Note that $C_S \leq C$ and hence the PSI values lie between 0 and 1. A high PSI value means that the amount of similarity preserved by function call indirections is also high. For example, if PSI $= 1$, then without function call indirections, all the identical function call statements in the old and new images use different target addresses, whereas with function call indirections they have same target addresses (except the ones in the indirection table). Note that PSI $= 0$ means that even without function call indirections, the identical call statements would have same target addresses because the locations of the corresponding functions are not changed by the software modification. Hence, in this case, there is no advantage due to Zephyr's function call indirections. Figure 13 shows the PSI values for different software change cases discussed in Section 8. For Case 1 and Case G where only a single parameter in the software is changed, PSI $= 0$ as expected. Note that for most of the cases, PSI has a very high value. This suggests that most of the software change cases cause many functions to be shifted and hence without function call indirections, the number of identical function call statements with different target addresses in the two versions of the software is high. This causes the delta script to be large. This explains the observation that byte-level comparison alone is not sufficient and application-level modifications are necessary to create a small delta script.

As shown in Figure 14, let us consider two code segments, one in the old and one in the new new version of the software, which are *identical* except that the target addresses of n identical function call statements are different. Without function call indirections and metacommands, we need $(n + 1)$ COPY commands and n INSERT commands to describe the difference between these code segments. Thus the delta script for these two identical code segments is $(n + 1) * 7 + n * 7 = 14n + 7$ bytes long (since each COPY and INSERT command takes 7 bytes assuming that the target

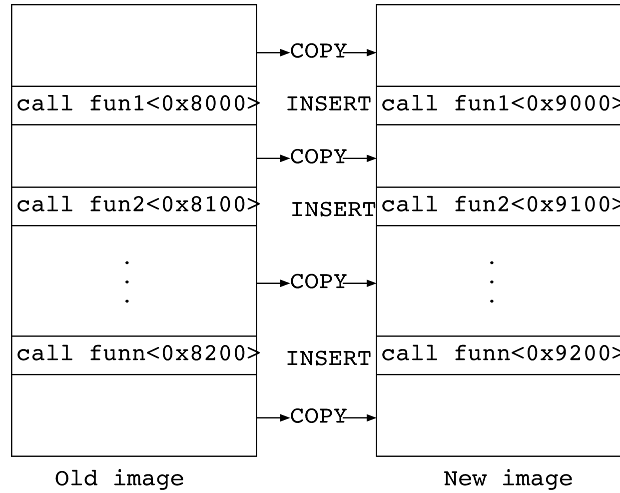


Fig. 14. Without function call indirections, the difference between the identical code segments require $n+1$ COPY commands and n INSERT commands in the delta script.

addresses of the function call statements is 2 bytes). With function call indirections, Zephyr needs only one COPY command which requires only 5 bytes (if we do not use *newOffset* as explained in Section 6.3). However, Zephyr also needs to describe, in the delta script, the difference between the corresponding segments in the indirection table. With the REPEAT command, this can be done with $3 + 2n$ bytes. Note that the CWI command is not necessary in this context. Thus Zephyr saves $12n - 1$ bytes compared to the scheme that uses only byte-level comparison without function call indirections and metacommands. In other words, function call indirections and metacommands decrease the size of the delta script by approximately 12 times the number of times the shifted functions are called in the new program code. To evaluate the effect of function call indirections only (without metacommands), let us assume that the baseline case takes 5 bytes each for COPY and INSERT commands (i.e., *newOffset* is not used). Then the delta script for these identical code segments takes $(n + 1) * 5 + n * 5$, whereas Zephyr takes 5 bytes for COPY command and $3 + 2n$ bytes for the REPEAT command. Thus function call indirections save $8n - 3$ bytes compared to the scheme that uses only byte-level comparison without function call indirections. Note that these savings are the worst-case figures, since the number of function references are generally higher than the number of functions. As a result, the REPEAT command needs less than $3 + 2n$ bytes in Zephyr.

10. CONCLUSIONS

In this article, we presented a multihop 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. We 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 that do not provide dynamic linking on the nodes (like the standard release of TinyOS), as well as to incrementally upload the changed modules in operating systems like SOS and Contiki that do 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 existing techniques. This leads to

reductions in reprogramming time and energy. As future work, we are investigating the use of multiple nodes as the source of the new code instead of a single base node to further speed up reprogramming.

REFERENCES

BROUWERS, N., LANGENDOEN, K., AND CORKE, P. 2009. Darjeeling, A feature-rich vm for the resource poor. In *Proceedings of the ACM 7th International Conference on Embedded Networked Sensor Systems (Sensys'09).*

BUETTNER, M., YEE, G., ANDERSON, E., AND HAN, R. 2006. X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proceedings of the ACM 4th International Conference on Embedded Networked Sensor Systems (Sensys'06).*

CC2420. Cc2420 homepage. http://focus.ti.com/lit/ds/symlink/cc2420.pdf

CZAJKOWSKI, G. 2000. Application isolation in the Java virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00).* 354–366.

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

EL-HOIYDI, A. AND DECOTIGNIE, J. 2005. Low power downlink mac protocols for infrastructure wireless sensor networks. *Mobile Netw. Appl.*, 675–690.

ESTADIUM. eStadium homepage. http://estadium.purdue.edu

HAN, C., RENGASWAMY, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. 2005. SOS: A dynamic operating system for sensor networks. In *Proceedings of the ACM International Conference on Mobile Systems, Applications and Services (MobiSys'05).* 163–176.

HUI, J. AND CULLER, D. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the ACM SIGOPS International Conference on Embedded Networked Sensor Systems (SenSys'04).* 81–94.

INC, C. 2003. Mote in-network programming user reference. http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf

JARSYNC. jarsync homepage. http://jarsync.sourceforge.net/

JEONG, J. AND CULLER, D. 2004. Incremental network programming for wireless sensors. In *Proceedings of the IEEE International Conference on Sensor and Ad Hoc Communications and Networks (SECON'04).* 25–33.

KAMRA, A., MISRA, V., FELDMAN, J., AND RUBENSTEIN, D. 2006. Growth codes: Maximizing sensor network data persistence. In *Proceedings of the ACM SIGCOMM Data Communications Festival.* 255–266.

KLUES, K., LIANG, C., PAEK, J., MUSALOIU-E, R., LEVIS, P., TERZIS, A., AND GOVINDAN, R. 2009. TOSThreads: Thread-Safe and noninvasive preemption in TinyOS. In *Proceedings of the ACM 7th International Conference on Embedded Networked Sensor Systems (Sensys'09).*

KOSHY, J. AND PANDEY, R. 2005a. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks.* 354–365.

KOSHY, J. AND PANDEY, R. 2005b. VMSTAR: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the ACM SIGOPS International Conference on Embedded Networked Sensor Systems (SenSys'05).* 243–254.

KRASNIEWSKI, M., PANTA, R., BAGCHI, S., YANG, C., AND CHAPPELL, W. 2008. Energy-Efficient on-demand reprogramming of large-scale sensor networks. *ACM Trans. Sensor Netw. 4,* 1.

KULKARNI, S. AND WANG, L. 2005. MNP: Multihop network reprogramming service for sensor networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05).*

LEVINE, J. 2000. *Linkers and Loaders*. Morgan Kaufmann.

LEVIS, P. AND CULLER, D. 2002. Maté: A tiny virtual machine for sensor networks. *ACM SIGOPS Oper.g Syst. Rev.,* 85–95.

LEVIS, P., GAY, D., AND CULLER, D. 2005. Active sensor networks. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI'05).*

LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. 2003. TOSSIM: Accurate and scalable simulation of entire tinyOS applications. In *Proceedings of the ACM SIGOPS International Conference on Embedded Networked Sensor Systems (SenSys'03).* 126–137.

LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. 2004. Trickle:A self regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI'04).* 15–28.

MARRON, P., GAUGER, M., LACHENMANN, A., MINDER, D.AND SAUKH, O., AND ROTHERMEL, K. 2006. FLEXCUP: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN'06).* 212–227.

MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. 2001. A low-bandwidth network file system. In *Proceedings of the SIGOPS Symposium on Operating Systems Principles (SOSP'01).* 174–187.

PANTA, R. AND BAGCHI, S. 2009. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom'09).*

PANTA, R., KHALIL, I., AND BAGCHI, S. 2007. Stream: Low overhead wireless reprogramming for sensor networks. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (InfoCom'07).* 928–936.

PANTA, R., KHALIL, I., BAGCHI, S., AND MONTESTRUQUE, L. 2008. Single versus multi-hop wireless reprogramming in sensor networks. In *Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom'08)*. 1–7.

POLASTRE, J., HILL, J., AND CULLER, D. 2004. Versatile low power media access for wireless sensor networks. In *Proceedings of the ACM 2nd International Conference on Embedded Networked Sensor Systems (Sensys'04).* 95–107.

PUCHA, H., ANDERSEN, D., AND KAMINSKY, M. 2007. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the ACM Symposium on Networked Systems Design and Implementation (NSDI'07).*

REIJERS, N. AND LANGENDOEN, K. 2003. Efficient code distribution in wireless sensor networks. In *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'03).* 60–67.

SERRANO, M., BORDAWEKAR, R., MIDKIFF, S., AND GUPTA, M. 2000. Quicksilver: A quasi-static compiler for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'00).* 66–82.

STATHOPOULOS, T., J., H., AND D., E. 2003. A remote code update mechanism for wireless sensor networks. CENS Tech. rep.

TINYOS. homepage. http://www.tinyos.net

TRIDGELL, A. 1999. Efficient algorithms for sorting and synchronization. PhD thesis, Australian National University.

XBOW. homepage. http://www.xbow.com

Received July 2009; revised January 2010; accepted May 20101