

Incremental Network Programming for Wireless Sensors

Jaemin Jeong
EECS Department
University of California, Berkeley
Berkeley, California 94720
jaemin@eecs.berkeley.edu

David Culler
EECS Department
University of California, Berkeley
Berkeley, California 94720
culler@cs.berkeley.edu

Abstract—We present an incremental network programming mechanism which reprograms wireless sensors quickly by transmitting the incremental changes for the new program version. Using the Rsync algorithm we generate the difference of the two program images, which allows us to distribute just the key changes of the program. Unlike previous approaches, our design does not assume any prior knowledge of the program code structure and can be applied to any hardware platform. To meet the resource constraints of wireless sensors we tuned the Rsync algorithm which was originally made for updating binary files among computationally powerful machines. In our design, the sensor node processes the delivery and the decoding of the difference script in separate steps. This makes it easy to extend for multi-hop network programming. We are able to achieve the speedup of 9.1 for changing a constant and 2.1 to 2.5 for changing a few lines in the source code over the non-incremental delivery.

I. INTRODUCTION

Typically, wireless sensors are designed for low power consumption and small form factor and don't have enough computing power and storage to support the rich programming environment. Thus, the program code is developed in a more powerful host machine and is loaded to a sensor node. The program code is usually loaded to a sensor node through the parallel or serial port that is directly connected to the host machine. This is called in-system programming (ISP). In-system programming is the most common way of programming sensor nodes because it is supported by most microcontrollers, but it loads the program code to only one sensor node at a time. The programming time increases proportional to the number of wireless sensors to be deployed.

Network programming transmits the program code to multiple nodes over the wireless link. And this saves the efforts of programming each individual sensor node. XNP (Crossbow Network Programming) is the network programming implementation introduced with TinyOS 1.1 release [1], [2]. It supports basic function of code distribution and reprogramming, but it does not support multi-hop delivery and incremental update which is necessary for programming a large sensor network in a faster way.

We present an incremental network programming mechanism which sends the new version of program by transmitting the difference of the two program images. Unlike previous approaches, we generate the program code difference by

comparing the program code in block level without any prior knowledge of the program code structure. This gives a general solution that can be applied to any hardware platform.

We use the Rsync algorithm [10] to generate the difference. The Rsync algorithm finds the shared code blocks between the two program images and allows us to distribute just the key changes of the program. Originally, the Rsync algorithm was made for computationally powerful machines exchanging the update of binary files over a low-bandwidth communication link. We tuned the Rsync algorithm for wireless sensor network programming.

First, we made expensive operations, such as building the hash table, be processed by the host program in favor of a resource constrained sensor node. The sensor node simply reads or writes code blocks to the flash memory to rebuild the program image.

Second, we structured the difference to avoid unnecessary flash memory accesses. In rebuilding the program image, the sensor node processes the script dissemination and the decoding in separate steps. This makes it easy to use other dissemination protocols and to extend for multi-hop network programming.

We are able to achieve speedup of 9.1 for changing a constant and 2.1 to 2.5 for changing a few lines in the source code over the non-incremental delivery.

In Section II, we describe the concept of network programming. In Section III, we review related work. We present our incremental network programming mechanism and results in Section IV and V. We extend this for more efficient delivery in Section VI and conclude in Section VII.

II. BACKGROUND

In general, network programming is processed in three steps: (1) encoding, (2) dissemination and (3) decoding (Fig. 1). In the first step, the host program reads the application program code and prepares code packets to disseminate. In the second step, the host program sends the code packets. And the sensor nodes store the code packets in the storage space after receiving them. Finally, the network programming module rebuilds the program code and calls the boot loader to transfer the program code to the program memory.

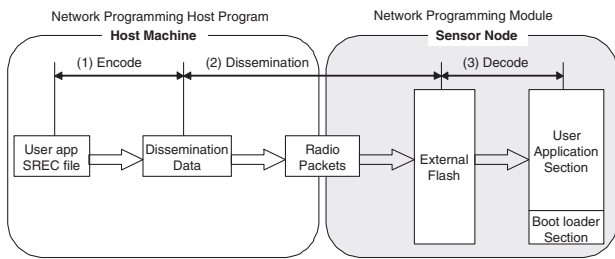


Fig. 1. Process of network programming

We will explain this in context of XNP to make things more clear. In encoding step, the host program simply reads the program code and stores each line of the program code in an array.

In dissemination step, the host program sends the program code as radio packets and the sensor nodes store the packets in the external flash memory after receiving them. The packets are written to the external flash memory because the network programming module is running in the user level and cannot write the program code directly to the program memory. Sensor nodes request the retransmission of any missing packets to the host program.

The code packets written in the external flash memory is the same format as the original program code. In decoding step, the sensor nodes just verify the program image and call the boot loader to transfer the program code to the program memory. The boot loader resides in the boot loader section of the microcontroller and has the privilege to write data bytes to the user application section of the program memory. After the boot loader finishes copying the program code, it restarts the system and the newly built program begins execution.

III. RELATED WORK

As mentioned in the previous section, any network programming mechanism consists of three steps: encoding, dissemination and decoding. We can compare different network programming schemes by focusing on these three steps.

XNP [1], [2] is the network programming implementation for TinyOS 1.1 release. It broadcasts the whole program code in a single hop network.

MOAP [4] is a multi-hop network programming implementation. One of the challenges of multi-hop network programming is to propagate the program code to multiple sensor nodes without saturating the network. MOAP uses an algorithm called Ripple dissemination protocol to disseminate the program code packets to a selective number of nodes without flooding the network. For packet retransmission, MOAP uses sliding window protocol. Sliding window protocol allows a sensor node to forward the program code while it is waiting for the retransmission of the lost packets.

Deluge [5] is a multi-hop network programming mechanism and it has two main functions: dissemination of block data and reprogramming boot loader. Deluge disseminates the program code in an epidemic fashion to propagate the program code while regulating the excess traffic. In order to increase the

transmission throughput, Deluge uses optimization techniques like adjusting packet transmission rate and spatial multiplexing. Unlike MOAP, Deluge uses a fixed sized page as a unit of buffer management and transmission.

Reijers et al [3] developed an algorithm that can efficiently encode the program code update. To describe the difference between the two program codes, the host program generates “edit script” which consists of operations like copy, insert, address repair and address patch. These operators modify the program code at the instruction level. Using complex operations like address repair and address patch helps reduce the network traffic, but it increases EEPROM accesses. One drawback is that their work just includes the encoding scheme and does not show a fully functional network programming implementation. Another drawback is that their algorithm is a processor specific solution. When we are going to apply their algorithm to a hardware platform with different processor, we have to redesign the operations because operations are dependent on specific instruction set.

Kapur et al [6], [7] implemented an incremental network programming based on Reijers et al’s encoding protocol and MOAP multi-hop network programming protocol. Their implementation supports the incremental multi-hop delivery of native code, but it has a limitation similar to Reijers et al’s algorithm. It is dependent on specific processor instruction set and does not provide a general solution.

The implementations above transmit the program code in native code. Whereas Maté [8] transmits the virtual machine code which is an application specific code for Maté virtual machine. One advantage is that a sender node doesn’t need to send the network programming module because it is assumed that the virtual machine is already running on the receiver node. This allows Maté to distribute program code quickly. One drawback of Maté is that it executes only the virtual machine instructions and a regular sensor application should be converted to the virtual machine instructions before execution.

Trickle [9] is an improvement over Maté. In Maté, each sensor node distributes the code by flooding the network with packets. This can lead to the network congestion and the algorithm cannot be used for a large sensor network. Trickle uses an epidemic algorithm to propagate the program code only to the sensor nodes that needs to be modified.

Our implementation supports incremental delivery of the native code. Unlike previous approaches, we generate the program code difference by comparing the program code in block level without any prior knowledge of the program code structure. This gives a general solution that can be applied to any hardware platform.

Table I summarizes the different network programming schemes for wireless sensor network.

Outside the sensor network community, there have been efforts to update program code incrementally. Rsync [10] is a mechanism that efficiently synchronizes the remote copy of an arbitrary binary file over a low-bandwidth, bidirectional communication link. Rsync finds any shared blocks between the two files. If we naively compare the blocks of the two files

TABLE I
COMPARISON OF NETWORK PROGRAMMING SCHEMES

	Encoding / Decoding	Dissemination
XNP	native code	Single hop
MOAP	native code	Multi-hop
Deluge	native code	Multi-hop
Reijers et al	Incremental native code (processor specific)	N/A
Kapur et al	Incremental native code (processor specific)	Multi-hop (MOAP)
Maté Trickle	virtual machine code	Multi-hop
This paper	Incremental native code (processor neutral general solution)	Single hop

at each byte position, the cost of comparison would be high. Rsync addresses this problem by having two levels of hash (checksum, hash). To compare two blocks, the algorithm first compares the checksum values of the two blocks. Only when the checksums match, the Rsync algorithm compares the hash value to ensure the correct match.

IV. DESIGN AND IMPLEMENTATION

To design an incremental network programming mechanism, we need to consider some factors that affect the performance. Compared to other sensor applications, network programming keeps a large amount of data in sensor nodes. Since the programming time is proportional to the data size, reducing the amount of transmission will improve the programming time. The program code is stored in the external memory before it is copied to the program memory, but the external flash memory is much slower than the on-chip memory. For better performance, the access to the external flash memory should be made only when it is necessary. When we divide the role of sensor nodes and the host program, we want sensor nodes to process only the key operations in an inexpensive way because sensor nodes are much more constrained than the host program in terms of processing power.

We present an incremental network programming mechanism in two stages. In section IV and V, we focus on finding shared data between the two program versions. In section VI, we will extend this for more efficient delivery.

A. Storage Organization

XNP stores the program image in a contiguous memory chunk in the external flash memory. We can extend this by allocating two memory chunks, one for the previous program image and the other for the new program image that will be built (Fig. 2). This memory organization has an advantage that it provides the same view of the memory as XNP and minimizes the effort of rewriting the boot loader code. The boot loader code of XNP reads the program code assuming that it is located at a fixed location in the external flash memory. We modified the boot loader so that it reads the program code from the base address value passed by the network programming module.

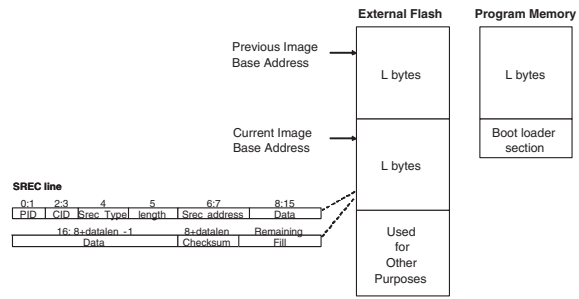


Fig. 2. External flash memory allocation

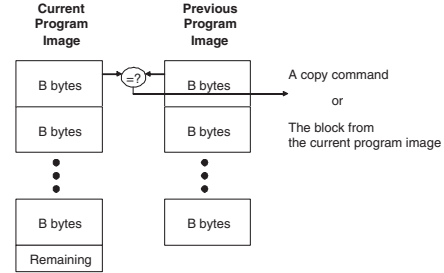


Fig. 3. Generating difference with Fixed Block Comparison

B. Difference Generation

For the incremental network programming, one might think of a simple extension of XNP like this: The host program generates the program difference by dividing the program image files into fixed size blocks and comparing the corresponding blocks of the previous and the new versions (Fig. 3). We call this scheme as Fixed Block Comparison. Fixed Block Comparison is not effective because it cannot find the shared blocks when the program code image is shifted.

Instead, we use the Rsync algorithm [10] to generate the difference and rebuild the program image. The Rsync algorithm was originally made for efficiently updating binary data between the two machines over a low-bandwidth network. It works as follows: (1) The sender node asks the receiver the version of blocks the receiver has. (2) The receiver sends the list of checksums for its blocks. (3) The sender compares the latest version with the checksums and sends the difference. (4) The receiver rebuilds the latest version using the previous version and the difference.

This algorithm is advantageous in that the sensor nodes don't have to keep track of receiver's version history, but can be problematic if applied to wireless sensors as it is. The sensor nodes have to calculate the list of checksum pairs and this requires expensive operations like external flash memory scan and MD-4 hash computation. Since wireless sensors are usually under the control of the system administrator, we can assume that the host program knows the version history of the sensor nodes. Then the expensive checksums can be calculated locally in the host program. And the sensor nodes just rebuild the program image by writing the code blocks. In case the host program does not know the version history, it can program the

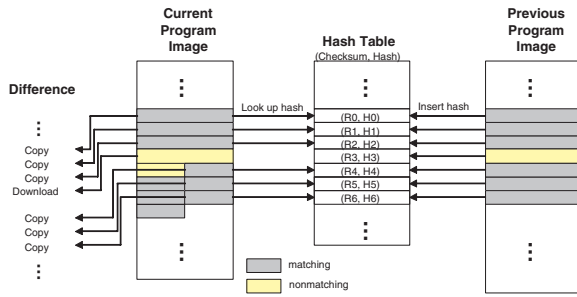


Fig. 4. Rsync difference generation

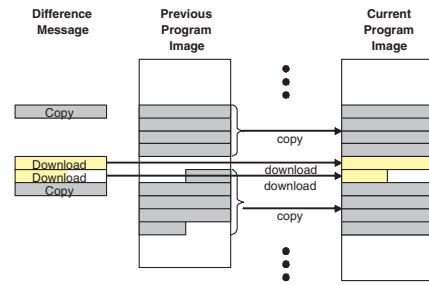


Fig. 5. Downloading a non-matching block

sensor nodes in non-incremental mode.

Using the Rsync algorithm, the host program generates the difference as in Fig. 4.

- 1) The Rsync algorithm calculates a checksum pair (checksum, hash) for each fixed sized block (e.g. B bytes) of the previous program image. And the checksum pair is inserted into a hash table for look-up.
- 2) Rsync reads the current program image and calculates the checksum for the B byte block at each byte. If it finds a matching checksum in the hash table, Rsync calculates the hash for the block and compares it with the corresponding entry in the hash table. If the hash also matches, then the block is considered a matching block.
- 3) Rsync moves to the next byte for comparison if the block doesn't have a matching checksum or a hash. A region of bytes that doesn't have any matching blocks is tagged as non-matching block and needs to be sent explicitly for rebuild.

Fig. 4 illustrates how the Rsync algorithm captures a matching block. Suppose the program image is shifted in the middle due to a modification operation. Rsync forms a B byte window and calculates the hash for it. If the modified bytes are different from any blocks in the previous program image, then the hash of the modified bytes doesn't match any hash table entry with very high probability. Rsync moves the window one byte at a time and calculates the checksum for any possible match. It doesn't match until Rsync starts to read unmodified blocks. At this moment, Rsync has found a matching block.

We used Jarsync [11] for the Rsync algorithm implementation on the host.

C. Program Rebuild

The host program describes the difference as two operations: 'copy' (CMD_COPY_BLOCK) and 'download' (CMD_DOWNLOADING). The host program sends a copy message for each matching block in the difference. After hearing the message, the sensor node copies the block in the previous image to the current image. For each non-matching block in the difference, the host program sends one or multiple download messages.

The block size of a copy message is set as a multiple of SREC line size and the block size of a download message

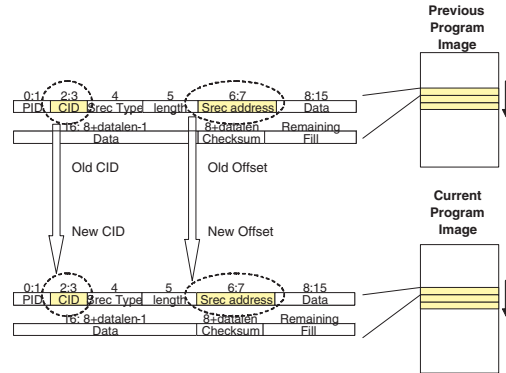


Fig. 6. Copying a matching block

is arbitrary number. The host program divides a download message block into multiple fragments when a download message block is bigger than an SREC line size (16 bytes).

Rebuilding the program image is straightforward when the structure of the program does not change: The sensor nodes just write download blocks and execute copy operations. When the program image is shifted, it gets a little bit complicated. Suppose the block is shifted by k from the original location of the previous image and k is not a multiple of SREC line size (Fig. 5). Then, the block can straddle between the two SREC lines and this can cost two SREC line writes. To avoid unnecessary SREC line writes, we used the following techniques:

- 1) When a download data block fragment is smaller than the SREC line size, the sensor nodes write the fragment as an SREC line without filling the remaining bytes. The length field of the SREC is set as the actual line size (Fig. 5).
- 2) The copy data block that comes after the non-multiple size download data block is written at the SREC line boundary. This ensures that a line of the copy data block fits within an SREC line. And the SREC address field of the SREC line is modified to the offset within the new program image (Fig. 6).

D. Code Complexity

To estimate the complexity of our implementation, we counted the source code lines in XnpM.nc file. A CMD_DOWNLOADING message costs 136 lines and a

```

// Case 1: Changing Constants
command result_t StdControl.start() {
  // Start a repeating timer that fires every 1000ms
  return call Timer.start(TIMER_REPEAT, 1000);
}

// Case 2: Modifying Implementation File
event result_t Xnp.NPX_DOWNLOAD_DONE
(uint16_t wProgramID, uint8_t bRet,
 uint16_t wEENofP){
  if (bRet == TRUE)
    call CntControl.start();
  // these two lines are added
  else
    call CntControl.stop();
  return SUCCESS;
}

// Case 4: Commenting out IntToLeds component
configuration XnpCount {
}
implementation {
  components Main, Counter, /*IntToLeds,*/
  IntToRfm, TimerC, XnpCountM, XnpC;
  ...
  //Main.StdControl->IntToLeds.StdControl;
  //IntToLeds<-Counter.IntOutput;
  ...
}

```

Fig. 7. Test Scenario

CMD_COPY_BLOCK message (for Rsync) costs 153 lines. These numbers are comparable to those of other TinyOS modules (Table II).

TABLE II
COMPLEXITY OF INCREMENTAL NETWORK PROGRAMMING

Incremental Network Programming		Radio Stack MAC		ADC Operation
Download	Copy (Rsync)	Send	Receive	Get and DataReady
136	153	112	88	35

V. EVALUATION OF BLOCK DISTRIBUTION

A. Experiment Setup

To evaluate the performance of this design choice, we will count the number of block or packet transmissions of the test set. We consider the following five cases as a test scenario (Fig. 7):

- Case 1 (Changing Constants) : This is the case with the minimum amount of change. We modified the constant in XnpBlink that represents the blinking rate of the LED. XnpBlink is an application written for demonstrating the network programming.
- Case 2 (Modifying Implementation File) : This is a more general case of program modification. We added a few lines of code to XnpCount program. XnpCount is a simple network programmable application. It counts a number, displays the number in its LEDs and broadcasts the number in radio packets.
- Case 3 (Major Change) : In this case, we generate the difference of the two programs XnpCount and XnpBlink. The difference is bigger than the first two cases.

TABLE III
TRANSMISSION TIME FOR FIXED BLOCK COMPARISON

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.6KB
SREC lines	1139	1167	1167	1156	1155
L_{down}	19	911	1135	1124	1123
L_{copy}	1120	256	32	32	32
N_{copy}	70	16	2	2	2
Estimation					
T (ms)	23280	114120	136800	135480	135360
T_{xnp} (ms)	136680	138720	140040	138720	138600
Speed-up (T_{xnp}/T)	5.87	1.22	1.02	1.02	1.02
Measurement					
T (ms)	25094	124403	149044	147149	146848
T_{xnp} (ms)	149888	152996	152996	150477	150465
Speed-up (T_{xnp}/T)	5.97	1.23	1.03	1.02	1.02

- Case 4 (Modifying Configuration File) : We commented a few lines in XnpCount program so that we do not use IntToLeds module. IntToLeds is a simple module that takes an integer input and displays it in LEDs of the sensor node.
- Case 5 (Modifying Configuration File) : We commented a few lines in XnpCount program so that we do not use IntToRfm module. IntToRfm takes an integer input and transmits it over the radio packet.

B. Results

To evaluate the performance, we estimated and measured the transmission time. The host program calculates the estimated transmission time by counting the following metrics:

- L_{down} : number of SREC records (16 byte data) sent by download messages.
- L_{copy} : number of SREC records to be copied by copy messages.
- N_{copy} : number of copy messages.

If it takes t_{down} to send a download message and t_{copy} to send a copy message, then the transmission time for Fixed Block Comparison, T , can be calculated as follows:

$$T = L_{down} \cdot t_{down} + N_{copy} \cdot t_{copy}$$

As a baseline for comparison, we can also calculate the transmission time for the non-incremental delivery as follows:

$$T_{xnp} = L_{down} \cdot t_{down} + L_{copy} \cdot t_{down}$$

As for t_{down} and t_{copy} , we found right values after a number of trials. We set these as 120 ms and 300 ms respectively for MICA2 sensor nodes.

1) *Fixed Block Comparison*: Table III shows the estimation and measurement data for Fixed Block Comparison. Since very few packets were lost (less than 2 packets for each case), we counted just the transmission time for the measurement data not including the retransmission time.

In case 1, the difference between the two program images is small. Most SREC lines (1120 out of 1139) are transferred by copy messages and the speedup (T_{xnp}/T) is about 5.9.

In case 2, where we added a few lines in the source code, we find that less than a quarter of the SREC lines are transferred by copy messages (256 out of 1167) and the speedup is 1.2.

In case 3, only 32 out of 1167 lines are transferred by copy messages and the speedup is about 1.03. Even though XnpBlink and XnpCount share much in source code level, they have little sharing in binary code level. The main reason is that XnpCount uses the radio stack components while XnpBlink does not. The radio stack is one of the most important modules in TinyOS and it takes a number of source code lines.

In case 4 and 5, where we commented out IntToLeds and IntToRfm components in the configuration file XnpCount.nc, we find that only a small number of lines are transferred by copy messages and the speedup is very small (1.02 for each case).

Fixed Block Comparison is not so effective for the incremental network programming. It works well when the program structure doesn't change (case 1). But, the level of sharing is low when we added a few lines of code (case 2), which we think a more general case of program modification.

2) *Using the Rsync algorithm:* Table IV and Fig. 8 show the results for the implementation using the Rsync algorithm.

TABLE IV
TRANSMISSION TIME WITH THE RSYNC ALGORITHM

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.2KB	49.4KB	49.4KB	48.9KB	48.9KB
SREC lines	1120	1154	1156	1140	1147
L_{down}	4	200	888	326	871
L_{copy}	1116	954	278	814	276
N_{copy}	72	104	85	107	83
Estimation					
T (ms)	22080	55200	132060	71220	129420
T_{xnp} (ms)	134400	138480	139920	136800	137640
Speed-up (T_{xnp}/T)	6.09	2.51	1.06	1.92	1.06
Measurement					
T (ms)	23812	61015	142607	77090	140314
T_{xnp} (ms)	148823	148889	148889	148172	148016
Speed-up (T_{xnp}/T)	6.25	2.44	1.04	1.92	1.05

In case 1, most SREC records (1116 lines out of 1120) were transferred and the speedup over the non-incremental delivery is 6.25 (measurement).

In case 2, 954 lines out of 1154 lines were transferred by copy messages and the speedup over the non-incremental delivery is 2.44 (measurement).

In case 3, the level of sharing is much smaller and the speedup is 1.04 (measurement). We have some number of copy messages (85 messages), but they cover only a small number of blocks and are not so helpful in reducing the programming time.

In case 4, 814 lines out of 1140 lines were transferred by copy messages and the speedup over the non-incremental delivery is 1.92 (measurement).

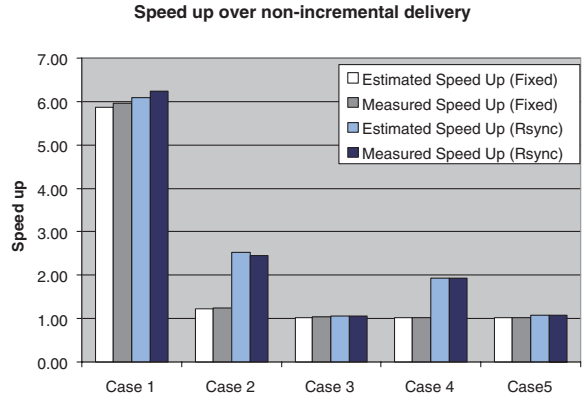


Fig. 8. Comparison of programming time

In case 5, 276 lines out of 1140 lines were transferred by copy messages and the speedup over the non-incremental delivery is quite small – 1.06 (measurement). We commented out a few lines both in case 4 and 5. But, in case 5, commenting out IntToRfm component made the radio stack not used and this changed the layout of the program image file a lot.

In summary, using the Rsync algorithm achieves the speedup of 6 for changing the constant and 2.4 for adding a few source code lines. These numbers are bigger than those of Fixed Block Comparison, but using the Rsync algorithm is not still effective with the major code change.

VI. OPTIMIZING DIFFERENCE DELIVERY

Compared to naive Fixed Block Comparison, using the Rsync algorithm reduces the programming time by efficiently finding the shared blocks between the two binary code files. However, we can find something to improve:

First, the data block size of a copy message is limited to the external flash memory page size (256 bytes). A copy message with the long data block is split to multiple copy messages whose block size fits within a flash memory page. This is to bound the processing time of a copy message so that the network programming module finishes processing a copy message before it receives another message. However, this increases the communication time. Case 1 of Table IV exemplifies this: The host program sent 72 copy messages (N_{copy}) even though only 4 SREC lines (L_{down}) are different between the two program images.

Second, the network programming module interprets a copy message immediately without queuing it. In case there is a missing command, the network programming module has to check the rebuilt program image because it hasn't stored the script commands. Since the network programming module does not know whether a missing hole is caused by a missing copy message or a number of download messages, it sends a retransmission request for each missing record in the current program image. This will take more time than retransmitting only the missing command.

We extend the implementation of Section IV as follows:

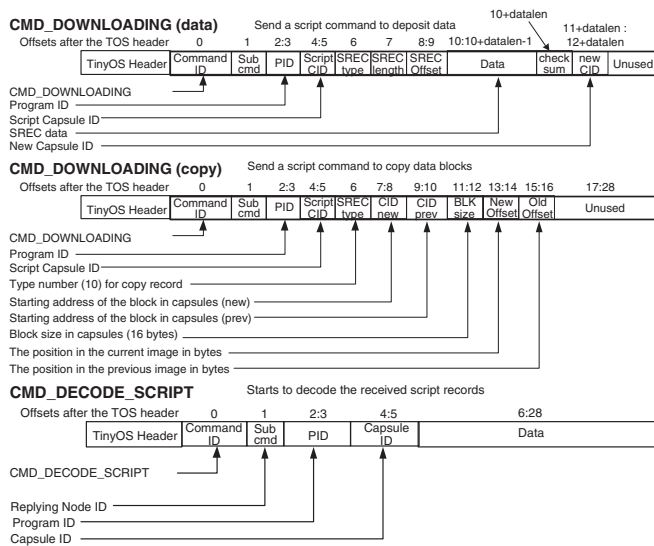


Fig. 9. Message format for Rsync with decode

- 1) The sensor node queues all the script commands.
- 2) The sensor node checks any missing records for the script.
- 3) The sensor node decodes the script after receiving the decode message.

A. Design

1) *Operations:* To send the script commands, the host program embeds each script command in a CMD_DOWNLOADING message. The embedded command is stored in the script queue of the external flash memory in a similar way to data download commands and we can reuse most of the code to process script commands (Fig. 9).

To embed a data download message, we changed the meaning of the CID (SREC line number) field. CID field now represents the SREC line number in the script queue. The SREC line number where the data block will be written at the decode stage is saved in the new CID field.

A copy message is also embedded in a CMD_DOWNLOADING. To differentiate an embedded copy message from other message types, we used a special value for the SREC type field. The SREC specification allows only several values for this field (0,1,2,3,5,7,8 and 9). We extended the meaning of this field so that the value 10 represents an embedded copy message. This allows us to store a copy command in the same way as other data records, but still interprets the copy command correctly.

CMD_DECODE_SCRIPT message makes the network programming module start decoding the script commands.

2) *Storage Organization and Program Rebuild:* As for the storage space for the script commands, we use the external flash memory because the external flash memory has enough space even for a very long script. We divide the external flash memory into three sections: the previous program image, the current program image and the script queue.

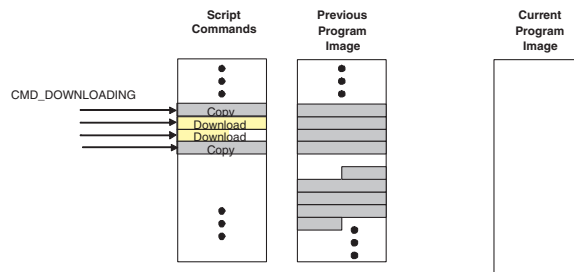


Fig. 10. Receiving script commands

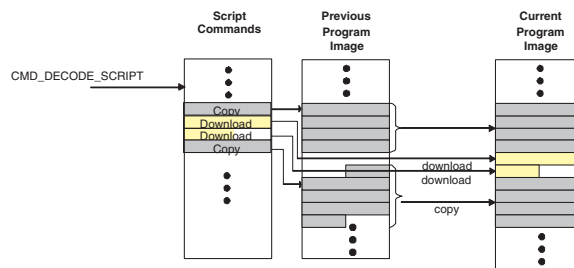


Fig. 11. Decoding script commands

At first, the host program sends the script as ‘download’ (CMD_DOWNLOADING) messages and the sensor node stores these messages in the script queue (Fig. 10). When the host program queries any missing script commands, the sensor node scans the script queue. If the sensor node finds any missing record, it requests the retransmission of the record. Then, the host program sends the record again.

The sensor node starts decoding the script after it receives the decode command from the program (Fig.11). The sensor node extracts the download or copy command embedded in the script command and interprets the command accordingly. The embedded commands are not exactly the same format as SREC records and need to be modified a bit before being processed. For a download command, the sensor node writes the SREC record in the current program image section by filling the CID (SREC line number) field with the real CID value (Fig. 9). For a copy command, the sensor node copies SREC lines after changing the CID and the byte offset fields.

B. Results

Since a sensor node rebuilds the program image in two steps (script transmission and decoding), we modified the metrics for the evaluation. We measured the transmission and the decode time for the five cases (Table V).

For case 1, only 7 script messages were transmitted and this made the transmission time very small. The sum of transmission time and the decode time is 16015 ms while it takes 154043 ms for the non-incremental delivery. This gives the speedup of 9.10. For case 2, more script lines were transmitted (337 script messages for the 1167 line program code) and the speedup over the non-incremental delivery is 2.53. For case 3, we sent even larger number of script messages (996 messages for 1167 line program code) and the

TABLE V

TRANSMISSION TIME WITH THE RSYNC ALGORITHM AND SCRIPT DECODE

	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.7KB
SREC lines	1139	1167	1167	1156	1156
# of commands	7	337	996	419	964
Measurement					
T (ms)	922	45843	130653	54544	125577
T_{decode} (ms)	16015	16687	16874	16765	16796
T_{xnp} (ms)	154043	158481	158481	150654	150525
Speed-up $T_{xnp}/(T+T_{decode})$	9.10	2.53	1.07	2.11	1.06

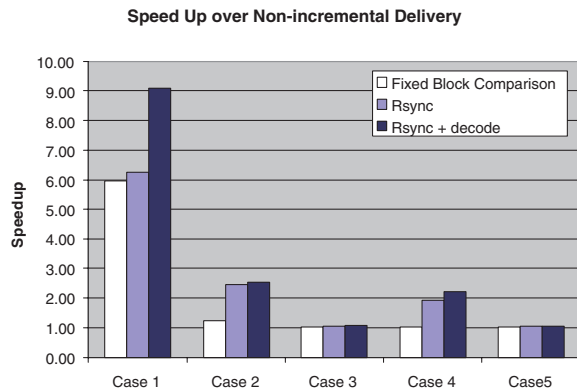


Fig. 12. Comparison of programming time

speedup was 1.07. When we modified the configuration file, we had the similar results with the Section IV. For case 4, 419 script messages for the 1156 line program code and the speedup over the non-incremental delivery is 2.11. For case 5, most of the SREC records were transmitted as download script commands (964 out of 1156) and the speedup was 1.06.

Fig. 12 and Table VI show the results of the three incremental network programming implementations: Fixed Block Comparison, Rsync and Rsync with split decode. We can find that splitting the script transmission and the program rebuild improves the overall programming time. When the source code is modified at minimum, the implementation with Rsync and split decode saves programming time by sending fewer script messages even though it has to decode the script messages. When a small number of source code lines are added, the programming time is a little better than the implementation that just uses the Rsync algorithm. For the major program change, it doesn't achieve the speedup, but it is still as good as the non-incremental delivery.

We can comment on case 3. Even though we used the Rsync algorithm and split decode, the speedup over the non-incremental delivery was negligible. This is because the difference between the two program images cannot be described with a small number of insert, copy and skip operations.

TABLE VI

COMPARISON OF PROGRAMMING TIME

Fixed Block Comparison					
	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.6KB
T (ms)	23280	114120	136800	135480	135360
T_{xnp} (ms)	136680	138720	140040	138720	138600
Speed-up (T_{xnp}/T)	5.87	1.22	1.02	1.02	1.02
Rsync					
	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.2KB	49.4KB	49.4KB	48.9KB	48.9KB
T (ms)	22080	55200	132060	71220	129420
T_{xnp} (ms)	134400	138480	139920	136800	137640
Speed-up (T_{xnp}/T)	6.09	2.51	1.06	1.92	1.06
Rsync with split decode					
	Case 1	Case 2	Case 3	Case 4	Case 5
File size	48.9KB	50.1KB	50.1KB	49.7KB	49.7KB
T (ms)	922	45843	130653	54544	125577
T_{decode} (ms)	16015	16687	16874	16765	16796
T_{xnp} (ms)	154043	158481	158481	150654	150525
Speed-up $T_{xnp}/(T+T_{decode})$	9.10	2.53	1.07	2.11	1.06

C. Efficient Dissemination

To disseminate the script commands, we used the network protocol of XNP. The host program sends a script command by embedding each script command in a download message. This implementation reprograms sensor nodes faster than XNP by transmitting the key changes of the program code, but it has something to improve:

- 1) It does not reprogram a sensor node which is more than one hop from the host program.
- 2) It does not fully utilize the space in the command packets because it sends each command as a single packet.

We can use efficient data dissemination protocol like Deluge to address these problems. Deluge supports multi-hop delivery and transmits any data assuming as byte streams. Since our implementation processes encoding, dissemination and decoding separately, it is straightforward to apply a different dissemination protocol like Deluge. For this, we need to convert the data format between byte stream and array of commands.

VII. CONCLUSION

Network programming is a way of programming wireless sensor nodes by sending the program code over radio packets. By sending program code packets to multiple sensor nodes with a single transfer, the network programming saves the programming efforts for a large sensor network.

We extended the network programming implementation of TinyOS 1.1 release so that it reduces the programming time by transmitting the incremental update rather than the whole

program code. The host program generates the difference of the two program images using the Rsync algorithm and transmits the difference to the sensor nodes. Then, the sensor nodes decode the difference script and build the program image based on the previous program version and the difference script. We tested our incremental network programming implementation with some test applications. We have achieved the speedup of 9.1 for changing a constant and 2.1 to 2.5 for changing a few lines of code in the source code over the non-incremental delivery.

REFERENCES

- [1] Jaemin Jeong, Sukun Kim and Alan Broad, "Network Reprogramming," TinyOS document, <http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf>
- [2] Crossbow Technology, "Mote In Network Programming User Reference," TinyOS document, <http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf>
- [3] Niels Reijers and Koen Langendoen, "Efficient Code Distribution in Wireless Sensor Networks," *WSNA '03*
- [4] Thanos Stathopoulos, John Heidemann and Deborah Estrin, "A Remote Code Update Mechanism for Wireless Sensor Networks," *CENS Technical Report # 30*, <http://lecs.cs.ucla.edu/~thanos/moap-TR.pdf>
- [5] Adam Chlipala, Jonathan Hui and Gilman Tolle, "Deluge: Data Dissemination in Multi-Hop Sensor Networks," *UC Berkeley CS294-1 Project Report, December 2003*, http://www.cs.berkeley.edu/~jwhui/research/projects/deluge/deluge_poster.ppt
- [6] Rahul Kapur, Tom Yeh and Ujjwal Lahoti, "Differential Wireless Reprogramming of Sensor Networks," *UCLA CS213 Project Report, December 2003*
- [7] Tom Yeh, Haru Yamamoto and Thanos Stathopoulos, "Over-the-air Reprogramming of Wireless Sensor Nodes," *UCLA EE202A Project Report, December 2003*, http://lecs.cs.ucla.edu/~thanos/EE202a_final_writeup.pdf
- [8] Philip Levis and David Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *ASPLOS Oct. 2002*
- [9] Philip Levis, Neil Patel, Scott Shenker, and David Culler "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*.
- [10] Andrew Tridgell, "Efficient Algorithms for Sorting and Synchronization," *PhD thesis, Australian National University, 1999*
- [11] Casey Marshall, "Jarsync: a Java implementation of the rsync algorithm," <http://jarsync.sourceforge.net/>

ACKNOWLEDGMENT

The authors would like to thank Crossbow Technology for providing the source code for the network programming module and the boot loader. This work was supported in by DARPA NEST Contract F33615-01-C1895 and California Energy Commission Demand Response Enabling Technology Development Project.