Node-level Representation and System Support for Network Programming

Jaein Jeong
Department of Electrical Engineering and Computer Science
University of California, Berkeley

December 16, 2003

Abstract

One problem of the current network programming implementation in TinyOS 1.1 is that it takes much longer than the tranditional in-sytem-programming due to the slow radio connection. We extended the implementation so that it uses the history of the previous version to reduce the transmission time. For a small amount of change in the source code, we achieved around 3.5 times of the speed-up. And for the case where no code blocks were shared, our implementation worked as much as the current implementation.

Keywords: network programming, sensor networks

1 Introduction

A typical wireless sensor node doesn't have enough computing power and storage to support the rich programming developement environment. Thus, the program code is developed in a more powerful host machine (e.g. PC) and is loaded to a sensor node later typically with in-system-programming (ISP); the program code is loaded to a sensor node through the parallel or serial cable that is directly connected to the host machine. Since most microcontrollers support program loading with the parallel or serial port, ISP is the most common way of programming sensor nodes. However, ISP can be a problem when a large number of sensor nodes need to be deployed because the program delivery time increases with the number of sensor nodes with ISP. Modifying the program source code for bug fixes or new functionalities can delay the program development cycle a lot. In addition, ISP involves all the efforts of collecting the sensor nodes placed in different locations and possibly disassembling and reassembling the enclosures. Network programming can be used to address these problems. With network reprogramming, the program code is sent to one or more of sensor nodes through the radio channel without any wiring between sensor nodes and the host machine (Figure 1).

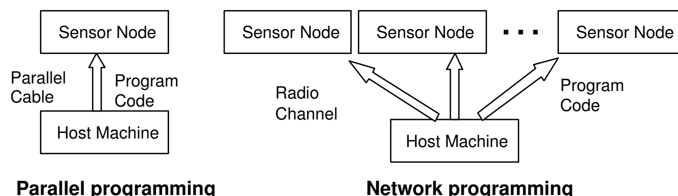

Figure 1: Comparison of network reprogramming with parallel programming.

In the network reprogramming, the program code is dissemnated as radio packets and stored in the external storage (e.g. EEPROM). Then, the boot loader, copies the downloaded program code to the program memory and makes it available for execution. Network programming is supported in the latest wireless sensor platform of UC Berkeley (TinyOS 1.1 with MICA2 and MICA2DOT hardware platform).

One of the flaws of the current implementation is that it is not optimized for fast delivery. All the program code is sent even though the change between the current version and the previous one is small. Our project goal is to achieve fast code delivery by reducing the number of transmissions without incurring much overhead on the sensor node.

2 Background

2.1 Concepts

Once the source code for a wireless sensor node is successfully compiled, the binary code is generated and available for loading. In UC Berkeley sensor platform which we used for this work, the souce code is written in nesC programming language. The source code is compiled to the binary code (`main.exe`) after several intermediate steps. This file is further converted to Motorola SREC format (`main.srec`) which is an ASCII representation of binary code to make the postprocessing easier. Each line of an SREC file contains the data bytes of the binary code with additional house keeping information (Figure 2).

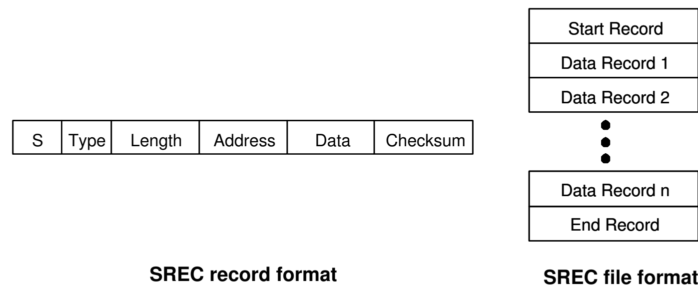

Figure 2: Format of SREC file and its records.

In ISP, a host programming tool is used to load the program code to the microcontroller through the direct connection to the host machine, for example, the parallel port. The host programming tool sends a special sequence of signals that leaves the microcontroller in the programming mode. For UC Berkeley sensor platform, UISP is used as a host programming tool. UISP starts ISP by setting $\overline{RESET}$ to 0, which makes Atmega128 microcontroller in MICA2 or MICA2DOT sensor node go to the programming mode. Then, UISP reads each record of the SREC file and sends it to the microcontroller. After that, the microconroller writes the received data bytes in its program memory (Figure 3).

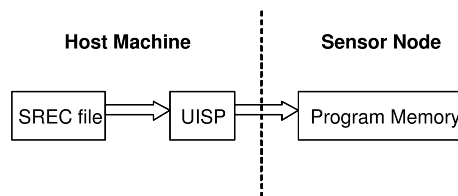

Figure 3: Process of parallel programming

Network programming takes a different approach to load the program code. A sensor node cannot be in the programming mode and receive the program code as in ISP because the sensor node is not connected to the host machine. Instead, the program code is loaded to the sensor node in two phases using the user level network programming module and the boot loader (Figure 4(a)). In the first phase, the network programming module stores the program code in the storage outside the program memory. In the second phase, the boot loader copies the program code in the external storage to the program memory. Since the network programming module is in the user application section, it does not have the write access to the program memory for memory protection reasons. Instead, it writes the program code in the external flash outside the program memory. The network programming module requests any missing records of the program code to make sure that thre are no missing records. After that, it calls the boot loader. The boot loader resides in the boot loader section in the high memory area of the ATmega128 microcontroller. The boot loader has the privelege to write data bytes to the user application section of the program memory. After copying the program code stored in the external flash to the program memory, it restarts the system.

The network programming module and the boot loader initially need to be installed in a sensor node because they are not built-in to the sensor node (Figure 4(b)). The user application wired to the network programming module and the boot loader are installed using UISP in a similar way to other applications.

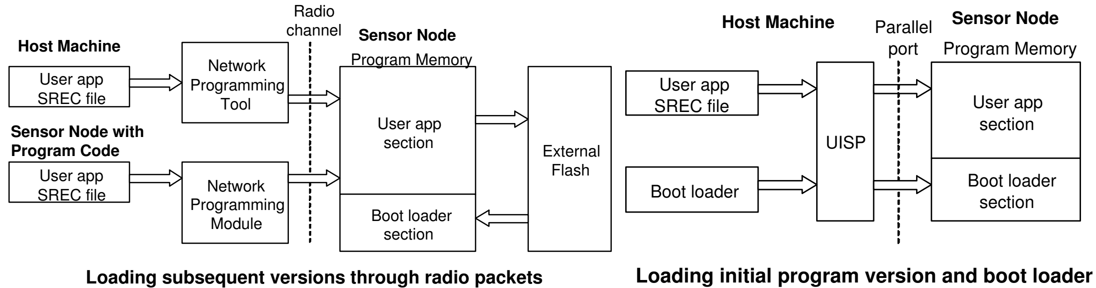

Figure 4: (a) Process of network programming, (b) Installation of initial version and the boot loader

2.2 Network Programming in TinyOS 1.1

Our work is based on the network programming implementation in TinyOS 1.1 release [3], [4]. This implementation was written by Crossbow Technology and was modified by UC Berkeley. We will call this implementation as XNP (Crossbow Network Programming). XNP consists of three components: the network programming module, the boot loader and the network programming host tool.

The network programming starts with code delivery. First, the user selects the program code to send in XNP network programming host tool. The host tool reads the program code and generates the 16-bit hash of the code (program ID) to identify this version of the code. If the user selects 'Download', then the host program starts code delivery by sending 'Download Start' messages (Figure 5(a)). In response to the 'Download Start' message, the XNP network programming module saves the program ID and sets the pointer to the external flash as the starting address. In addition, it asks the user application whether it will start network programming. This can be used to disable network programming when the system is busy or the network programming is not needed.

After sending the 'Download Start' message in several tries, the host program transmits the SREC file by sending multiple 'Download' messages (Figure 5(b)). A 'Download' message contains one line from the SREC file as its data, program ID and line number in the SREC file (CID: capsule ID). The network programming module on the sensor node receives the 'Download' message and writes it in the external

flash memory. Once the download stage finishes, the host program sends 'Download Terminate' message to the sensor nodes.

Before loading the newly downloaded code, the sensor nodes should make sure that none of the SREC lines are missing. The host program sends 'Query' messages to the sensor nodes after it finishes sending 'Download' messages and 'Download Terminate' message. In response, the network programming module reads through the external flash memory storage. It checks the validity of each SREC line by reading the program ID and the sequence number. If it finds any missing hole, it replies to the host program for the missing line. Then, the host program sends an 'Update' packet for the line and the sensor nodes writes the recieved packets in the external flash memory. If a sensor node has no more missing holes, it also replies to the host program. The host program sends 'Update' packets until it doesn't hear any responses from the sensor nodes.

The next step is to copy the program code in the external flash to the program memory. If the user selects 'ReProgram' in the host program, the host program sends 'ReProgram' message. Then, the network programming module checks whether the program ID of the downloaded command matches the program ID in the 'ReProgram' message. If the program ID is correct, the network programming module calls the boot loader by jumping to the boot loader section. The boot loader reads each external flash line (32 bytes) from the starting address and copies the lines in a page size (256 bytes) to the program memory until it reads the end record. After that, it reboots the system by turning on the watchdog timer.

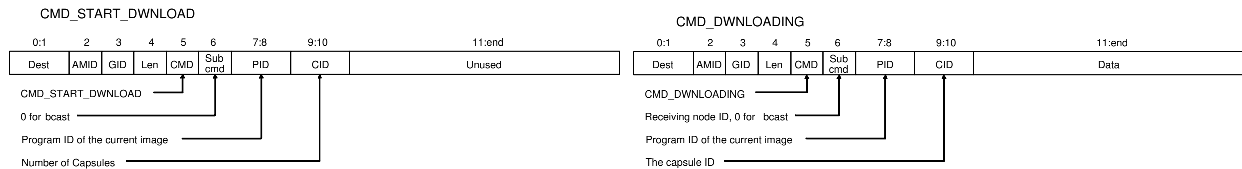

Figure 5: Start Download Message

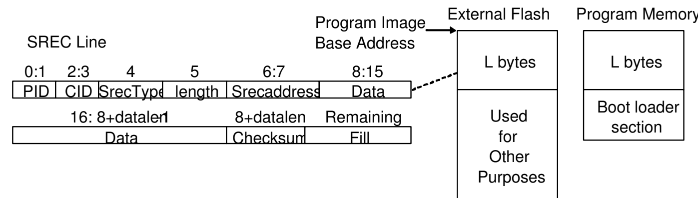

Figure 6: Writing an SREC line to the external flash

3 Design

Network programming implementation in TinyOS 1.1, XNP, does not consider the history of the previous version of program code. It transmits all the records of the SREC file each time it sends the program code. In XNP, the program image is stored in the beginning of the external flash memory as in Figure 6.

3.1 Memory Layout

Supporting Multiple Versions:

If we are to extend the network programming so that it transfers the difference instead of the whole program code, the previous program image should reside in the external flash memory. In order to maintain multiple versions of program code, we divided the external flash memory as multiple contigous chunks, each of which is as big as the program memory (Figure 7). For example, MICA2 and MICA2DOT nodes can have up to 4 chunks. This is because Atmega128 microcontroller of MICA2 and MICA2DOT platforms have 128KB of program memory and the external flash memory has 512KB (= 4Mbit) of space. To manage the data storage in the external flash memory, two pointers (the start of the current image and the start of the previous image) are used. This information is written in the Atmega128's non-volatile internal flash memory so that it exists even after the sensor node is powered off. The internal flash memory of the microcontroller has enough room (4KB) for keeping this information.

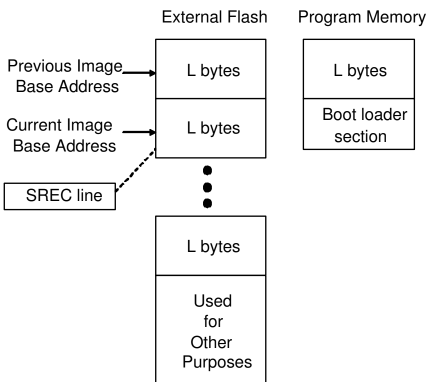

Figure 7: Modified Memory Layout

Generating Program Update:

When we send the new program code, we compare the new code with the previous one. The program code in the two sections are compared in fixed sized blocks (256 bytes). Using a fixed sized block makes it easy to maintain the data structure on the sensor nodes. Then, the two blocks from the current and the previous program image are compared sequentially. If the two matches, a 'Copy' message is sent that tells the network programming module in the sensor nodes to copy the block stored in the previous program image to the current one. If the two blocks are different, each line of the block is sent (Figure 8).

3.2 Protocol Extension

Assuming multiple versions of code are maintained in a sensor node, we extended the existing protocol.

Start Download:

Our network programming can send the new program code in the same way as XNP does or it can send the difference between the new version and the previous version which we call 'incremental download'. In incremental download, the 'Start Download' message specifies the program ID of the current image and the previous image and the number of records for the current image (Figure 9). The network programming module sets the pointers to the current section and the previous section by checking the program IDs.

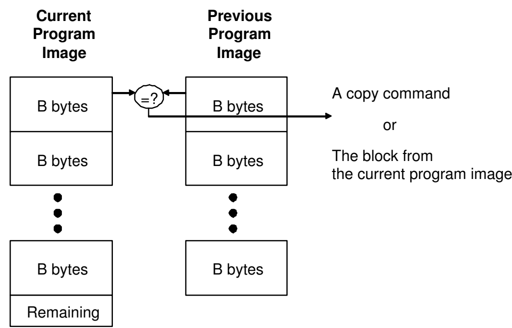

Figure 8: Generating Updates

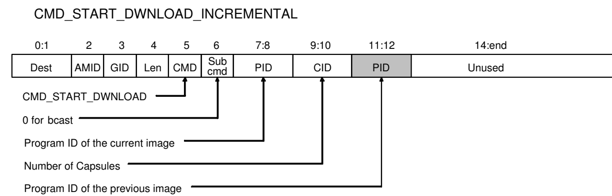

Figure 9: Modified Start Download

Download:

In XNP, the download message just sends a single line of SREC record. For incremental download, we have two operations, insert and copy, depending on whether the two blocks from the current program image and the previous one are the same or not. Copy operation tells the network programming module on the sensor node to copy the data bytes from the previous area to the current area.

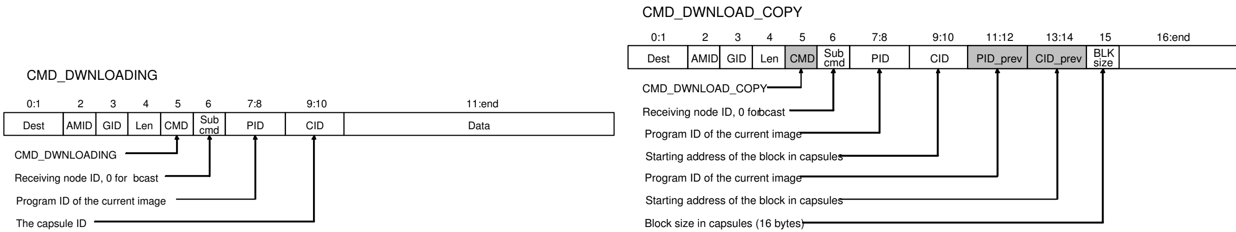

Figure 10: Modified Start Download

Query:

The format of query message and the reply message is the same as in XNP. However, the internal behavior has slightly changed to support multiple versions. The starting address to find the missing hole now points to the current program image section.

Reboot:

Similar to query stage, the format of reboot message is the same as in XNP. We made a small modification to the network programming module and the boot loader. In XNP, the network programming module

passes the beginning of the external flash memory as the first address to copy. But, the boot loader simply starts copying from the beginning of the external flash memory without using the parameter. In our extended version, the network programming module passes the beginning of the current program image section and the boot loader uses this parameter as the starting address to copy SREC records.

4 Evaluation

Setup

Since incremental network programming can reuse the common blocks in the code, we postulate better performance with small change of the program. For evaluation, we considered the following three cases:

- Case 1: To test the base case with the minimum program change, we generated two functionally equivalent program copies which are only different in the timestamp embedded during compilation time. As a test set, we used `XnpBlink`. `XnpBlink` is a simple network programmable application which blinks the red LED at 1 sec interval.
- Case 2: In this case, we modified a line in `XnpBlink` source code to change its LED blinking interval from 1 sec to a different value.
- Case 3: In this case, we sent the difference in the code for `XnpCount` from `XnpBlink` application.

`XnpBlink` and `XnpCount` are simple applications to demonstrate the use of network programming and their application specific code is small compared to the network programming code size (Table 1). Application specific code takes less than 10% of the total source code.

Table 1: Code size of test applications

	`XnpBlink`	`XnpCount`
# of source code lines for network programming modules	2049	2049
# of source code lines for application specific modules	157	198
# of SREC lines	1139	1166

The transmission time depends on how many blocks are common between the two programs because the host program compares in fixed sized blocks (16 lines = 256 bytes). Table 2 shows that most blocks are shared for case 1 and case 2. But, for case 3, most blocks are not common between the two program code.

Results

We ran experiments using real sensor nodes (MICA2). For the two network programming implementation (XNP and incremental network programming), we tested the three cases mentioned above. Table 3 and Figure 11 shows the results.

We timed the network programming for each step: start download, download, end download, query and reprogram stage. The timing difference are mostly from the download stage. The steps except the download took almost the same time for all cases either with XNP or the incremental network programming.

Table 2: Number of common blocks for each case

	Case 1	Case 2	Case 3
Lines in SREC file (previous image)	1139	1139	1139
Lines in SREC file (new image)	1139	1139	1166
Different Lines	1	2	1066
Common Lines	1138	1137	100
Blocks to transmit	1	2	71
Blocks to copy on the sensor node	71	70	2

Table 3: Results

Timing (sec)	Case 1		Case 2		Case 3	
	XNP	Incremental	XNP	Incremental	XNP	Incremental
Start/End Download	6.0	7.0	6.0	6.7	7.0	7.0
Download	149.0	25.0	148.7	27.0	152.0	148.7
Query	12.0	12.0	12.0	12.0	12.0	12.0
Reprogram	3.0	3.0	3.0	3.0	3.0	3.0
Total	170.0	47.0	169.7	48.7	174.0	170.7

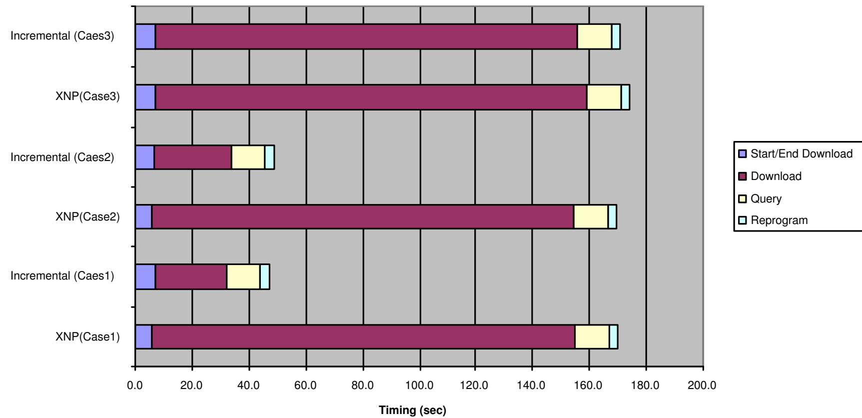

Figure 11: Results

For case 1 and 2, where most blocks between the two programs are the same, the incremental network programming ran much faster than XNP (3.6 time speed-up for case 1 and 3.5 for case 2). For case 3, where most blocks are not common, the programming time was close to that of XNP.

5 Related Works

Reijers et al [1] developed algorithms to reduce the size of an edit script to perform a program update. Besides simple copy and insert operations used in diff, they used address shifts and address patches to minimize the size of the edit script. Their algorithms were effective in reducing the communication traffic and this was used to evaluate their works. However, they didn't mention the non-communication part in the evaluation. Using address shifts and address patches put more overhead on the sensor node. When generating the new program code, they copied the old code rather than sharing the code. This can lead to the increased EEPROM access time and the waste of the memory.

Stathopoulos et al [8] developed Multihop Over-the-Air Programming (MOAP). They reduced the network traffic for network programming by using Ripple dissemination protocol with 60 - 90 % reduction of traffic compared to flooding. They also reduced the overhead of accessing EEPROM by using sliding window scheme. Their work is for sending the single program image and doesn't support incremental update.

While the two examples in the above are about deliverying the binary code in sensor networks, in Maté [10], the code is distributed over the radio channel in the form of virtual mahcine instructions. Trickle [9] improved the communication traffic of Maté by propagating program code using epidemic algorithms.

6 Conclusion

The network programming is a way of loading program code into wireless sensor nodes by sending radio packets. It is advantageous in that it can save the efforts of programming by propagating the program code to multiple sensor nodes simultaneously without any physical wiring. One problem is that the network programming takes much longer than the traditional in-system-programming due to the slow radio connection. We extended the current implementation in TinyOS 1.1 so that uses the history of the previous version to reduce the number of transmissions. For the case where we made a small change in the source code, we achieved around 3.5 times of speed-up. And for the case where almost no code blocks were shared, our implementation worked as much as the current implementation.

Discussion and Future Works

Our incremental code distribution was helpful reducing code propagation traffic for small change of code, but it could not utilize the commonalities for more general cases. We find this is because the network programming module, which is shared by the network programmable applications, is not placed in the beginning of the source code. Thus, the shared code is shifted and cannot be shared. This can be fixed with the help of network programming aware compiler. We can place the network programming module within a fixed location in the code by inserting compiler directives and inlining function calls. The compiler can recognize the network programming module and determine its location in the topological ordering. Another method is to find the similarities at arbitary points using a fingerprint [11].

Currently, the program storage for the current program version and the previous one exist in separate copies. Even though this extension has the advantage of reusing much of existing boot loader code, there are two problems with this approach. First, there is an overhead of block copy. The part of the code shared by the two program versions should be copied from the previous version to the section of the current version. Since accessing external flash is an costly operation in terms of time and energy, sharing the code without copy is advantageous. Second, if the code is shared, then the user doesn't need to keep track of program code history on the sensor nodes because the code block is not necessarily associated with a

single program. A sensor node can save the external flash memory space and it can use the space for more number of program versions or for other data logging purposes. We expect the cost of sharing code to be maintaining the directory structure for each fixed sized code block. This will require a major change in the network progarmming module and the boot loader.

Our work is mostly focused on the code management for a single hop network. We plan to combine our work with Deluge [12] to support multi-hop network programming.

Acknowledgements

This work was funded in part by the DARPA NEST Contract F33615-01-C-1815.
We would like to thank Crossbow Technology for providing the source code for the network programming module and the boot loader, as well as David Culler who gave us valuable comments on this work.

References

[1] Niels Reijers and Koen Loangendoen, "Efficient Code Distribution in Wireless Sensor Networks," *WSNA '03*

[2] Alan Demers et al, "Epidemic Algorithms for Replicated Database Mangement," *PODC '87*

[3] TinyOS document, "Network Reprogramming," http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf

[4] TinyOS document, "Mote In Network Programming User Reference," http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf

[5] Atmel, "ATmega 128 Microcontroller Reference," http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf

[6] Atmel, "Atmel 4-mega bit Data Flash Reference," http://www.atmel.com/dyn/resources/prod_documents/doc1938.pdf

[7] B. Kang et al, "The Hash History Approach for Reconciling Mutual Inconsistency," *ICDCS '03*

[8] 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

[9] Philip Levis, Neil Patel, Scott Shenker and David Culler, "Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks," *UC Berkeley Tech Report UCB//CSD-03-1290, November 2003,* http://www.cs.berkeley.edu/~pal/pubs/trickle-techreport.pdf

[10] Philip Levis and David Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *ASPLOS Oct. 2002*

[11] Udi Manber, "Finding Similar Files in a Large File System," *Proceedings of Winter USENIX Conference (1994)*

[12] 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