Reprogramming with Minimal Transferred Data on Wireless Sensor Network

Jingtong Hu
Department of Computer Science
University of Texas at Dallas
Richardson, TX, USA
Email: jxh068100@utdallas.edu

Chun Jason Xue
Department of Computer Science
City University of Hong Kong
Tat Chee Ave, Kowloon, Hong Kong
Email: jasonxue@cityu.edu.hk

Yi He
Department of Computer Science
University of Texas at Dallas
Richardson, TX, USA
Email: yxh011010@utdallas.edu

Edwin H.-M. Sha
Department of Computer Science
University of Texas at Dallas
Richardson, TX, USA
Email: edsha@utdallas.edu

Abstract

In Wireless Sensor Networks, the preloaded program code and data on sensor nodes often need to be updated due to changes in user requirements or environmental conditions. Sensor nodes are severely restricted by energy constraints. It is especially energy consuming for sensor nodes to update code through radio packages. To efficiently update code through wireless radio, we propose an algorithm, Reprogramming with Minimal Transferred Data *(RMTD), to find the optimum combination of copying from the old code image and downloading from the host machine to minimize the number of bytes needed to be transferred from the host machine to a sensor node. Our experiments show that, for small code modifications, RMTD reduces the number of bytes transferred by 93.25% over the existing Rsync-based algorithm. For normal code changes, RMTD shows an improvement of 59.82% in average.*

1. Introduction

A Wireless Sensor Network(WSN) is a network that consists of many low-cost, battery-powered sensor nodes which are preloaded with application code and data. It is usually deployed into a field to track events of interest, such as wildlife habitat monitoring [1]. Since sensor nodes are usually left unattended after deployment, they are severely constrained by energy.

Due to the changes of user requirements and environmental conditions, the preloaded program code and data on wireless sensors often need to be updated. For example, a WSN may be deployed in unfamiliar territory. Using the information gathered, the code may need to be updated to find more interesting information. Reprogramming sensor nodes is more economical and practical than deploying new sensor nodes [2][3]. To disseminate new code through radio is energy intensive. Sending a single bit of data consumes about the same energy as executing 1000 instructions [4][5][6]. Thus, reducing data sent over radio extends the lifetime of sensor nodes. In this paper, we present the Reprogramming with Minimal Transferred Data (RMTD) algorithm to minimize the bytes transferred from the host machine to the sensor nodes to save power for the sensor nodes.

In general, network programming is processed in three steps: (1) encoding, (2) dissemination and (3) decoding. In the first step, a host program reads the application program code and prepares code packets to send. In the second step, the host program sends the code packets to sensor nodes. In the last step, the sensor nodes rebuild the program code. A simple way to update code in a sensor node is to send all of the new code image to the sensor node like Crossbow Network Programming(XNP) [7][8]. This is inefficient because there are usually many common segments between the old program image and the new program image. We can take advantage of this to reduce the number of bytes transferred. One way of reducing number of bytes sent is to compare the code of successive versions and generate an edit script that summarizes the differences [9][10]. This approach requires sensor node to interpret the script. Hence, hardware and performance overhead will increase. And this approach suffers the processor specific limitation. The other way of reducing number of bytes sent is to compress the code in the host machine and decompress it in the sensor node. This, too, is inappropriate because it will take too much power and hardware resources to conduct decompression in the sensor nodes.

Fixed-block comparison [11] was proposed to find common blocks between the old program image and the new program image. The host machine can send a small message to tell the sensor node to copy from old code image stored in its own memory to the new code image for each common

978-1-4244-5113-5/09/$25.00 ©2009 IEEE

block. This method needs less bytes to be sent than XNP, but it is not efficient when there is some shift in the code image. Jaein [12], tuned the Rsync algorithm [13], which was originally made for computationally powerful machines to update binary files over a low-bandwidth communication link. It can achieve some savings for minor changes in the source code, but for major code changes, it will not work effectively. Li et al. [14] considered the updating problem from the compiler's perspective of view. When compiling new code for sensor nodes, the compiler will, as much as possible, keep the new code the same as the old code. Their technique is orthogonal with our algorithm and can be combined with our algorithm to reduce data transfer further.

In this paper, we propose the RMTD algorithm to find common segments between the old code image and the new code image and compute the least number of bytes needed to be sent to sensor node to construct the new code image. The contributions of our paper are:

- RMTD algorithm takes advantage of partially constructed new code image in the sensor node besides old code image to reduce the data transfer.
- RMTD algorithm finds common segments in reverse order.
- With simple copy and download message, we prove that RMTD algorithm finds the least number of bytes to be transferred between the host machine and the sensor node.

The experimental results show that when there are small changes in the source code, we can reduce the number of bytes transferred by 93.25%. In average, we can reduce the number of bytes transferred by 59.82%.

Our paper is organized as follows: In Section 2, we present two motivational examples. The RMTD algorithm is presented in Section 3. Experimental results are provided in Section 4. Finally, we conclude our paper in Section 5.

2. Motivational Example

When updating the code of sensor nodes, we want to reduce the number of bytes transferred. Comparison-based algorithms are proposed to reduce number of bytes transferred to sensor nodes. Comparison-based algorithms compare the code of successive versions and find similarities and differences between the old code image and the new code image in the host machine. Then the host machine tells sensor nodes to either copy parts of the new code image from the old code image with a *copy message* or write parts explicitly with a *download message*. With the addition of the RMTD algorithm, there are three comparison-based algorithms based on code comparison.

1) The first algorithm is a fixed-block comparison(FBC) algorithm [11]. It divides the code images to blocks and then compare each corresponding block. If there is a matching block, it will send a copy message for the matching block. Otherwise, it will send the whole block via a download message.
2) The second algorithm is also a block-based algorithm [12]. It is based on the Rsync algorithm [13]. We call it the $Rsync_{berkeley}$ algorithm. It divides the old code image into blocks. It has a window with the same size as a block. This window goes through the new code image from the beginning, comparing the window with every block in the old code image. Whenever a match is found, a copy message is sent, and the window moves forwards a block. If a window does not match any blocks, then the window moves forward one byte, marking that byte as a non-matching byte. Once another match is found, it will send the non-matching bytes in a download message.
3) In the RMTD algorithm, we use a byte-oriented approach to find all common blocks between the old code image and the new code image. Common blocks in the RMTD algorithm are not of fixed-lengths. After we find all the common blocks between the old code image and the new code image, we use a dynamic programming approach to find the least number of bytes to be sent from the host machine to sensor nodes.

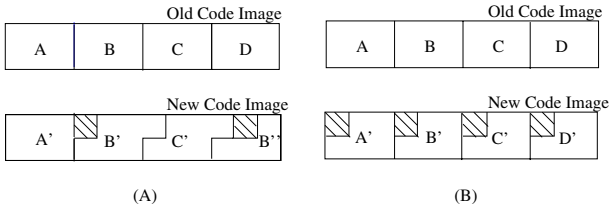

Figure 1. (A) : A new byte is inserted into the second block and the forth block is the same as the second block with last two bytes truncated. (B): The first byte of each block changes and D' is the same as D in a reverse order.

In this section, we provide two motivational examples to show that existing methods are not good enough in finding common blocks between the old code image and the new code image, and that we can do a better job in reducing the number of bytes transferred.

In our first example, which is shown in Figure 1 (A), our code images have four blocks. One new byte is inserted into the second block of the old code image. The third block is shifted one byte to the right accordingly. The forth block in the new code image has the same byte as previously inserted one inserted and the rest of this block is the same as the second block except that the last two bytes are truncated. This is possible when a function in the old code image is modified and called twice statically in the new code image.

We assume that a copy message takes 12 bytes and a download message takes the same number of bytes as bytes that need to be transferred. Furthermore, we assume that, in block-based algorithms, each block is 256 bytes.

Here are the results of the different algorithms running this example:

1) The FBC algorithm can find that block A and A' are common block, but the rest of these blocks will be sent via download messages explicitly. It sends one copy message and three download messages, totalling to 780 bytes.
2) $Rsync_{Berkeley}$ can find that blocks A, B, and C are the same as blocks A', B', and C'. It sends three copy messages for the common blocks and one download message for the last block, totalling to 292 bytes.
3) RMTD algorithm requires less bytes transferred. First it sends one copy message for block A, one download message for the inserted byte, and one more copy message for block B' and C'. (Note that, since RMTD is not block-based, it can copy a block of any length with one copy message.) At last, it will send a copy message to tell the sensor node to copy the inserted new byte and B'' from the already constructed new code. It sends Three copy messages and one 1-byte download message, totalling to 37 bytes.

In the second example, shown in Figure 1 (B), the first byte of each block changes and the forth byte D' is the same as D in a reverse order. The results of different algorithms running this example are as follows:

1) The FBC algorithm cannot find any common blocks. It sends the whole new code image to the sensor nodes via download messages. It sends a total of 1024 bytes.
2) $Rsync_{berkeley}$cannot find any common blocks either. It will also send total of 1024 bytes.
3) Since each block is the same as the old code image with the exception of the first byte, RMTD can find four common blocks. The forth byte in the new code image is the same as forth byte in the old code image in a reverse order, so we will have one copy message for this message. Totally, it sends a copy message for each of these blocks, and the four bytes in a download message. It sends a total of 52 bytes.

Table 1 summarizes the results of our examples. The last two columns show improvement of RMTD in this example over FBC and $Rsync_{berkeley}$ respectively.

Table 1. Example results.

	FBC	$Rsync_{berkeley}$	RMTD	Improvement	Improvement
Example 1	780	292	37	95.26%	87.33%
Example 2	1024	1024	52	94.92%	94.92%

3. RMTD Algorithm

In this section we first describe the communication model under which RMTD algorithm works. We then present our algorithm in detail, illustrating it with an example. Then we show the complexity of our algorithm. We prove that our algorithm minimizes the number of bytes to be transferred under the communication model.

3.1. Communication Model

We have an old code image X that sensor nodes need to update a new code image Y. Before the update, X is already on the sensor nodes, and host machine has both X and Y. The host machine wants to send the minimum number of bytes to the sensor nodes such that the sensor nodes can construct Y.

The host machine can send two types of messages: copy messages and download messages. Each copy message, consisting of the beginning address in X, the beginning address in Y, and the length of the segment to be copied, instructs the sensor node to copy a segment of X to Y. A download message is a segment of data that the sensor node writes into Y. With these two types of messages, sensor nodes can construct Y by copying of it from X, and downloading the remaining parts from the host machine.

In our model, each copy message will cost β bytes, and the cost of download messages will be the number of bytes sent.

3.2. Reprogramming with Minimal Transferred Data (RMTD)

Our algorithm is called *Reprogramming with Minimal Transferred Data (RMTD)*. The RMTD algorithm, as shown in Algorithm 3.1, consists of three phases:
Phase 1 finds all the common segments. Common segments include common segments between old code image and new code image and common segments between partially constructed new code image and the rest part of the new code image.
Phase 2 finds the optimal combination of common segments to copy, since there will be many overlaps between these common segments.
Phase 3 generates the copy and download messages. We use an example to illustrate our algorithm throughout this section.
RMTD operates on two binary files, and in this example, we use the alphabet to represent binary code. Each character represents one byte of binary code.

$X = \langle A, B, C, D, E, F, F, E, F, A \rangle$.
$Y = \langle D, E, F, A, B, C, A, F, E, F, E, D, C, B, A \rangle$.

Algorithm 3.1 Overall RMTD Algorithm

Input: Array of binary bytes of old code image X and array of binary bytes of new code image Y, X = $\langle x_1, x_2, ..., x_m \rangle$ and Y = $\langle y_1, y_2, .., y_m \rangle$

Output: Copy messages and download messages to be sent to the sensor nodes

1: Find Common Segments (Algorithm 3.4);
2: Find Optimal Combination of Copy and Download Messages (Algorithm 3.5);
3: Construct Messages (Algorithm 3.6);

Table 2. Mark the same bytes.

	j	0	1	2	3	4	5	6	7	8	9	10
i		Y	A	B	C	D	E	F	F	E	F	A
0	X											
1	D					z						
2	E						z			z		
3	F							z	z		z	
4	A		z									z
5	B			z								
6	C				z							
7	A		z									z
8	F							z	z		z	
9	E						z			z		
10	F							z	z		z	
11	E						z			z		
12	D					z						
13	C				z							
14	B			z								
15	A		z									z
16	A		\									z
17	B			\								
18	C				\							
19	D					\						
20	E						\			z		
21	F							\	z		z	
22	F								\		z	
23	E									\		
24	F										\	
25	A											\

3.2.1. Phase 1: Find Common Segments. Algorithm 3.4 takes array X, array Y, length of array X, length of array Y as inputs. First it will construct a table T. We put concatenation of X and Y in column and Y in row. If the i^{th} byte in column is the same as the j^{th} byte in the row, we will mark T[i,j] as 'z'. Table T for our example is shown in Table 2. Algorithm 3.4 calls Algorithm 3.2 and Algorithm 3.3. Algorithm 3.2 finds common segments that goes in same direction and Algorithm 3.3 finds common segments that goes in reverse direction. The outputs of Algorithm 3.4 consists of two parts. The first part is common segments between the old code image X and the new code image Y. The second part is common segments between partially constructed part of the new code image Y and the unconstructed part of Y.

After executing Algorithm 3.4, we obtain an array CS[ ]

Algorithm 3.2 Search Segment Forward

Input: Array X, Array Y, Position i in X, Position j in Y

Output: Starting position and ending position of common segment

1: **if** $i = -1$ or $j = -1$ **then**
2: **return** i+1, j+1 ;
3: **end if**
4: **if** $X[i] = Y[j]$ and T[i,j] = 's' **then**
5: $T[i, j] \leftarrow$'s';
6: Search Segment Forward $(X, Y, i-1, j-1)$;
7: **else**
8: **return** i+1, j+1 ;
9: **end if**

Algorithm 3.3 Search Segment Backward

Input: Array X, Array Y, Position i in X, Position j in Y

Output: Starting position and ending position of common segment

1: **if** $i = -1$ or $j =$ Length of Y **then**
2: **return** i+1, j-1 ;
3: **end if**
4: **if** $X[i] = Y[j]$ and T[i,j] = 's' **then**
5: T[i,j] $\leftarrow$ NULL;
6: Search Segment Backward$(X, Y, i-1, j+1)$;
7: **else**
8: **return** i+1, j-1 ;
9: **end if**

which records common segments. CS's format is (Starting in X, Ending in X, Starting in Y, Ending in Y) or (Starting in Y, Ending in Y, Starting in Y, Ending in Y). The result of Algorithm 3.4 applied to our example is: CS[ ] = {{(0, 2, 3, 5) forward between X and Y}, {(3, 5, 0, 2) forward between X and Y}, {(6, 8, 1, 3) forward between X and Y}, {(5, 4, 7, 8) backward between X and Y}, {(8, 0, 6, 14) backward between X and Y}, {(3, 1, 6, 8) backward between Y and Y}, {(2, 0, 9, 11) backward between Y and Y}, {(5, 3, 12, 14) backward between Y and Y}, {(7, 8, 9, 10) forward between Y and Y}}.

3.2.2. Phase 2: Find Optimal Combination of Copy and Download Messages . After obtaining array CS[ ], which records the common segments between X and Y, we begin to decide which segments are to be copied, and which segments are to be downloaded. If a segment is transferred by a download message, we say it is a *downloaded segment*. If a segment is transferred by a copy message, we say it is a *copied segment*.

We use our example to illustrate the complications caused by the overlap of common segments. There are totally 9 common segments after phase 1 in our example. Let us take a look at the first three common segments. The first common segment starts at 3 and ends at 5 in Y. The second common

Algorithm 3.4 Find Common Segments

Input: Array X, Array Y, Length of array X, Length of array Y
Output: Common segments

```
1: Construct Table T
2: for i = length[X] − 1 to 0 do
3:    for j = length[Y] − 1 to 0 do
4:       Call Algorithm 3.2 with (X, Y, i, j) as inputs;
5:       Store common segment between old code image
         and new code image in forward order;
6:    end for
7: end for
8: for i = length[Y] − 1 to 0 do
9:    for j = length[Y] − 1 to i do
10:      Call Algorithm 3.3 with (Y, Y, i, j) as inputs;
11:      Store common segment between constructed new
         code image and unconstructed new code image in
         forward order;
12:   end for
13: end for
14: for i = length[X] − 1 to 0 do
15:   for j = length[Y] − 1 to 0 do
16:      Call Algorithm 3.2 with (X, Y, i, j) as inputs;
17:      Store Store common segment between old code
         image and new code image in backward order;
18:   end for
19: end for
20: for i = length[Y] − 1 to 0 do
21:   for j = length[Y] − 1 to i do
22:      Call Algorithm 3.3 with (Y, Y, i, j) as inputs;
23:      Store common segment between constructed new
         code image and unconstructed new code image in
         backward order;
24:   end for
25: end for
```

segment starts at 0 and ends at 2 in Y. The third common segment starts at 1 and ends at 3 in Y. We use Figure 2 to illustrate these three segments. The rest of the common segments are not shown.

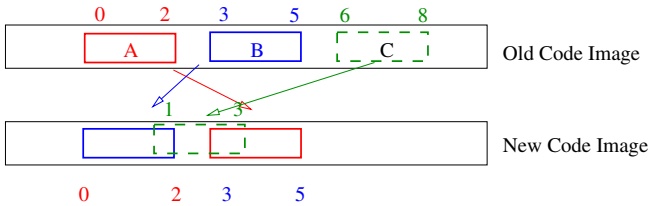

Figure 2. Example of overlapping segments.

Common segments are overlapping in Y, as shown in Figure 2. Such overlaps happen frequently when X and Y are large. If we copy all segments, we will waste resources. In Figure 2, if we copy all three common segments, bytes 1, 2, 3 will be copied twice. Actually, we only need to copy A and B. Thus, we need an algorithm to find the best combination of copy and download messages.

Given a set of common segments, Algorithm 3.5 finds the minimum number of bytes transferred such that sensor nodes can construct the new code image Y. Algorithm 3.5 employs a dynamic programming approach to accomplish this task.

Let $local\_optimum[i]$ be the minimum number of bytes transferred to tell sensor nodes how to construct Y up to and including the i^{th} byte of Y. Let $j(i)$ = $\min\{\ m\ |\exists\ a\ common\ segment(m,\ n), m \leq i \leq n\}$ where m and n are positions in Y. If such a common segment does not exist, we let $j(i) = \infty$. Our recursive formulation is shown in Equation 1.

We need two arrays $Sub[\ ]$ and $Message[\ ]$ to keep track of the information we need to construct copy and download messages. $Sub[i]$ stores the size of subproblem that has been solved, and $Message[i]$ stores the action to take to construct the last part of the new code image.

Algorithm 3.5 Find Optimal Combination of Copy and Download Messages

Input: Common segments array CS from Algorithm 3.4, Size of New code image N, Array Y
Output: Array s, Array Message, Array local_optimum

```
1: local_optimum[0] ← 0 ;
2: for i← 0 to N do
3:    local_optimum[i] ← local_optimum[i − 1] + 1 ;
4:    s[i] = i-1 ;
5:    Message[i] = Y[i] ;
6:    for k ← j(i) - 1 to i-1 do
7:       if local_optimum[i] ≥ local_optimum[k] + β
         then
8:          local_optimum[i] ← local_optimum[k] + β;
9:          Find corresponding address of k+1 in old code
            image from array CS, say l ;
10:         Sub[i] = k ;
11:         Message[i] = "Copy , Starting_X = l, Starting_Y
            = k+1, length = ( i - k )";
12:      end if
13:   end for
14: end for
```

Algorithm 3.5 first sets $local\_optimum[0]$ to be 0. Then it has a for loop to go from the first to the last byte of Y. For each byte i, it computes $local\_optimum[i]$. First, it sets $local\_optimum[i]$ = $local\_optimum[i-1]$ + 1 for the case when we download this byte. Then, it finds a common segment that intersects with i. If it will cost fewer number of bytes to send this byte with a copy message, it will set $local\_optimum[i]$ = $\min\limits_{j(i)-1 \leq k < i}\{\ local\_optimum[k] + \beta\}$.

$$local\_optimum[i] = \begin{cases} 0 & \text{if } i = 0 \\ local\_optimum[i-1] + 1 & \text{if } j(i) = \infty \\ \min(\min\limits_{j(i)-1 \le k < i}\{local\_optimum[k] + \beta\}, local\_optimum[i-1] + 1) & \text{if } j(i) \le i \end{cases} \quad (1)$$

3.2.3. Phase 3: Construct Messages. After arrays $Sub[\ ]$ and $Message[\ ]$ are obtained, we begin to construct the copy and download messages. The inputs to this phase are the arrays $Sub[\ ]$ and $Message[\ ]$, and the output are the copy and download messages. The initial invocation is $Construct\_Messages($ length of Y).

Algorithm 3.6 $Construct\_Messages$

Input: Arrays $Sub[\ ]$ and $Message[\ ]$
Output: Instructions for constructing copy and download messages

```
1: if i = 0 then
2:    return ;
3: else
4:    Construct_Messages(Sub[i]);
5:    Output Message[i];
6: end if
```

After host machine have all the messages, it sends to the sensor nodes.

Let n be the size of the code image. Since Algorithm 3.4 only visits each entry in table T once, and procedure $Search\_Segment$ in Algorithm 3.2 and Algorithm 3.3 will visit each entry in table T at most twice, the time complexity of Algorithm 3.4 is $\Theta(n^2)$. The complexity of Algorithm 3.5 is $\Theta(n^2)$. Hence, the complexity of the RMTD algorithm is $\Theta(n^2)$.

4. Experiments

To evaluate the performance of our algorithm, we compare the number of bytes transferred by RMTD with the number of bytes transferred by $Rsync_{berkeley}$. We use the source code of the dijkstra algorithm in the network package of Mibench[15] as the code we are going to update since dijkstra algorithm is widely used in sensor network routing algorithms.

For RMTD, we assume a copy message needs 4 bytes for the starting position in the old code image, 4 bytes for the starting position in the new code image, and 4 bytes for the length of the common segment. A download message costs the number of bytes to be sent to sensor nodes explicitly.For $Rsyc_{berkeley}$, we assume a copy message needs 4 bytes to indicate the sequence number of the block in the old code image. A download message will cost the number of bytes to be sent to sensor nodes explicitly. We assume one block contains 256 bytes.

The test cases are shown in Table 3. The first column show the difference between the old code image and the new code image, the second column show the file size of the new code image.

Table 3. Experiment test cases.

	Description	Binary File size
Case1	Change one constant	9328B
Case2	Add a counter	9340B
Case3	Change type of five variables	9404B
Case4	Add a variable	9460B
Case5	Change declaration of an array	9284B
Case6	Add a few lines of source code	9428B
Case7	Delete a few lines of source code	8960B
Case8	Comment a few lines of source code	9156B
Case9	Relocate a block of source code	9332B
Case10	Two totally different files	17020B

Table 4 shows the experiment results for $Rysnc_{berkeley}$ while Table 5 shows the experiment results for RMTD. In both tables, the first row shows the original file size. The second row shows how many bytes of the new code image is copied. The third row shows the ratio of copied bytes to the file size of the new code image. The forth row shows, totally, how many bytes are sent from the host machine to sensor nodes.

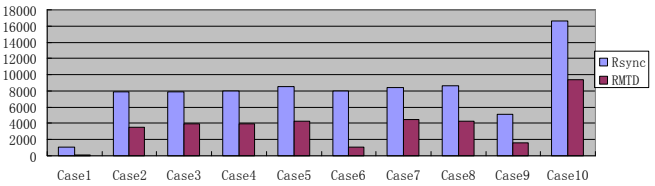

Figure 3. Comparison between $Rsync_{berkeley}$ and RMTD.

To evaluate the performance of our algorithm, we compare the total bytes transferred using $Rsync_{berkeley}$ and total bytes transferred using RMTD. The results are shown in Table 6. The first row shows how many bytes are transferred using $Rysnc_{berkeley}$ algorithm in each case, the second row shows how many bytes are transferred using our algorithm and the third row shows the percentage of improvement of our algorithm over $Rysnc_{berkeley}$ algorithm. Also, Figure 3 gives us a graphic illustration of the comparison between $Rsync_{berkeley}$ algorithm and RMTD algorithm.

The results show that when there are small changes in the source code, we can reduce the number of bytes transferred by 93.25%. In average, we can reduce the number of bytes transferred by 59.82%.

Table 4. Experiment result for $Rysnc_{berkeley}$ algorithm.

	Case1	Case2	Case3	Case4	Case5	Case6	Case7	Case8	Case9	Case10
Original File Size	9328B	9340B	9404B	9460B	9284B	9428B	8960B	9156B	9332B	17020B
Copied Bytes	8448B	1536B	1536B	1536B	768B	1536B	512B	512B	4352B	256B
Ratio	90.57%	16.45%	16.33%	16.24%	8.27%	16.29%	5.71%	5.59%	46.64%	1.50%
Total Bytes	1066B	7854B	7918B	7974B	8551B	7942B	8467B	8669B	5115B	16654B

Table 5. Experiment result for RMTD algorithm.

	Case1	Case2	Case3	Case4	Case5	Case6	Case7	Case8	Case9	Case10
Original File Size	9328B	9340B	9404B	9460B	9284B	9428B	8960B	9156B	9332B	17020B
Copied Bytes	9323B	8890B	8399B	8698B	8377B	8697B	7790B	8192B	9312B	7865B
Ratio	99.94%	95.18%	89.31%	91.94%	90.23%	92.25%	86.94%	89.47%	99.79%	46.21%
Total Bytes	72B	3529B	3939B	3911B	4221B	1056B	4464B	4299B	1546B	9350B

Table 6. Comparison.

	Case1	Case2	Case3	Case4	Case5	Case6	Case7	Case8	Case9	Case10
Use $Rsync_{berkeley}$	1066B	7854B	7918B	7974B	8551B	7942B	8467B	8669B	5115B	16654B
RMTD Algorithm	72B	3529B	3939B	3911B	4221B	1056B	4464B	4299B	1546B	9350B
Improvement	93.25%	55.07%	50.25%	50.95%	50.64%	86.70%	47.28%	50.41%	69.78%	43.86%

5. Conclusion

Along with fixed-block comparison algorithm, $Rysnc_{berkeley}$ algorithm, we propose a new RMTD algorithm to minimize the data transferred when reprogramming the sensor node. Our algorithm generates copy messages and download messages which will be sent to sensor node from host machine. Our algorithm can find the optimal combinations of copy messages and download messages so that the data transferred between host machine and sensor node is minimal. Our experiments result shows that when there are small changes in the source code, we can reduce the number of bytes transferred by 93.25%. In average, we can reduce the number of bytes transferred by 59.82%.

Acknowledgment

This work is partially supported by TI University Program, NSF CCR-0309461, NSF IIS-0513669, HK CERG B-Q60B and NSFC 60728206 and grants from City University of Hong Kong [Project No. 7200106][Project Number 9681001].

References

[1] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderso, "Wireless sensor networks for habitat monitoring," in *First ACM Int. Workshop on Wireless Sensor Networks and Application(WSNA)*, September 2002, pp. 78–89.

[2] Philip Levis and David Culler, "Mate: A tiny virtual machine for sensor networks," in *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, 2007.

[3] Philip Levis, Neil Patel, David Culler, and Scott Shenker, "Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *First USENIX/ACM Symposium on Networked System Design and Implementation(NSDI)*, 2004.

[4] Niels Reijers and Koen Langendoen, "Efficient code distribution in wireless sensor networks," in *International Workshop on Wireless Sensor Network Architecture*, 2003, pp. 60–67.

[5] Victor Shnayder, Mark Hempstead, Ror rong Chen, Geoff Werner Allen, and Matt Welsh, "Simulating the power consumption of large-scale sensor network applications," in *ACM Conference on Embedded Networked Sensor Systems(SenSys)*, 2004, pp. 188–200.

[6] Tom Yeh, Haru Yamamoto, and Thanos Stathopolous, "Over-the-air reprogramming of wireless sensor nodes," in *UCLA EE202A Project Report*, 2003.

[7] Jaein Jeong, Sukun Kim, and Alan Broad, "Network reprogramming," in *TinyOS document*.

[8] Crossbow Technology, "Mote in network programming user reference," in *TinyOS document*.

[9] Rajesh K. Panta, Issa Khalil, and Saurabh Bagchi, "Stream: Low overhead wireless reprogramming for sensor networks," in *IEEE Conference on Computer Communications(Infocom)*, 2007.

[10] Pedro Jose Marron, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel, "Flexcup: A

flexible and efficient code update mechanism for sensor networks," in *European Workshop on Wireless Sensor Networks*, 2006, pp. 212–227.

[11] Jaein Jeong, "Node-level representation and system support for network programming," Dec. 2003, p. 109.

[12] Jaein Jeong and David Culler, "Incremental network programming for wireless sensors," in *IEEE SECON*, Oct. 2004, p. 109.

[13] Andrew Tridgll, *Efficient Algorithms for Sorting and Synchronization*, Ph.D. thesis, Australian National University, 1999.

[14] Weijia Li, Youtao Zhang, Jun Yang, and Jiang Zheng, "Ucc:update-conscious compilation for energy efficiency in wireless sensor networks," in *PLDI'07*, Jun. 2007, pp. 104–109.

[15] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001.