Hermes- Fast and Energy Efficient Incremental Code Updates for Wireless Sensor Networks

Rajesh K. Panta, Saurabh Bagchi
Dependable Computing Systems Lab (DCSL)
School of Electrical and Computer Engineering, Purdue University

IEEE Infocom, 2009
Outline

• Introduction
• Zephyr
• Hermes
• Experiments
• Conclusion
Zephyr: Introduction

- Wireless reprogramming of sensor is an essential requirement since the software changes over time.

- It is essential to minimize the time to reprogram the network.

- The goal of reprogramming is to transfer a small delta (difference between the old and the new software)
Zephyr: Introduction

• Reprogramming
  • This necessitate modifying the executing application or retasking the existing application.
  • Reprogramming time and energy depend on the radio transmissions.
In this paper are as follows:

1) We create a **small-sized delta** for dissemination using optimized **byte-level** comparisons.

2) We design **application-level** modifications to increase the structural similarity between different software versions, also leading to **small delta**.
1) Zephyr performs **application-level** modifications on the old and new versions of the software to mitigate the effect of function shifts.
2) Then the two executables (old and new versions) are compared at the **byte-level** using a algorithm derived from the **Rsync** algorithm.
3) This produces the delta script which describes the difference between the old and new versions of the Software.
4) The **delta script** is transmitted to all the nodes in the network.

First the delta script is injected by the host computer to the base node
5) The nodes save the delta script in their external **flash** memory

After downloading the delta script, they rebuild the new image using and store it in the external flash
Zephyr: byte-level

• **Rsync algorithm** *(on the host computer)*
  • Rsync is an algorithm originally developed to update binary data between computers over a low bandwidth network.

  • Rsync divides the files containing the binary data into fixed size blocks.

  • Both sender and receiver compute the pair (Rolling Checksum, MD4) over each block.
Generate the pair (RollingChecksum, MD4) for each block of the old image and stores them into a hash table.

Calculate the checksum for the first block of the new image.

The algorithm checks if this checksum matches the checksum for any block in the old image by hash-table lookup.

Compare the MD4

The block is considered as a matching block

Move to the next block in the new image

Move to the next byte in the new image

The algorithm checks if this checksum matches the checksum for any block in the old image by hash-table lookup.

The block is considered as a matching block

Move to the next block in the new image

The block is considered as a matching block

Move to the next block in the new image

The block is considered as a matching block

Move to the next block in the new image
After running this algorithm, Zephyr generates a list of **COPY** and **INSERT commands** for matching and non matching blocks.

**COPY command:**
copies len number of bytes from `oldOffset` at the old image to `newOffset` at the new image.

**INSERT command:**
inserts `len` number of bytes `data` to `newOffset` of the new image.
Zephyr: byte-level

Rsync optimization

\[ x = y \]
\[ x + 1 = y + 1 \]

\[ x \sim x + 3 = z \sim z + 3 \]

\[ x = y = z \]
\[ x + 1 = y + 1 = z + 1 \]
Zephyr: byte-level

Rsync optimization

\[ x = y \]
\[ x + 1 = y + 1 \]
\[ x^{\sim}x + 3 = z^{\sim}z + 3 \]
\[ x = y = z \]
\[ x + 1 = y + 1 = z + 1 \]
Zephyr: byte-level

Rsync optimization

\[
\begin{align*}
x &= y \\
x + 1 &= y + 1 \\
x \sim x + 3 &= z \sim z + 3 \\
x &= y = z \\
x + 1 &= y + 1 = z + 1
\end{align*}
\]

We call contiguous matching blocks a super-block and the largest super-block the maximal super-block.
Zephyr: byte-level

Drawback of using only byte-level comparison.

**Case 1:**
Changing Blink application:
We change the application from **blinking LED every second** to **blinking it every 2 seconds**.

The delta script is **23 bytes**

**Case 2:**
We added just **4 lines of code** to Blink.

The delta script is **2183 bytes**.
**Zephyr : Application-level**

**Application-level modifications:**
As a result, all the calls to those **functions refer to new locations.**
This **produces several changes in the binary file** resulting in the large delta script.

```
| call fun1  | call fun1 |
| call fun2  | call fun2 |
| call funn  | call funn |
| a          | a'        |
| funn       | funn      |
| b          | b'        |
| fun1       | fun1      |
| c          | c'        |
| fun2       | fun2      |
| (a)        | (b)       |
```
Zephyr: Application-level

Function calls to the **indirection table**
Zephyr: Application-level

**Old image**

- Call loc1
- Call loc2
- Call loc3
- Call loc4

**New image**

- Call loc1
- Call loc5
- Call loc3
- Call loc6
- Call loc7

**Indirection table (new image)**

- loc1: Call fun1
- loc2: empty
- loc3: Call fun3
- loc4: empty
- loc5: Call fun7

**Indirection table (old image)**

- loc1: Call fun1
- loc2: Call fun2
- loc3: Call fun3
- loc4: Call fun4

**Delta size of Case II:**

2183 bytes => 280 bytes
Hermes: Introduction

We identify two problems related with Zephyr in particular and incremental reprogramming in general

1) Function call indirections **decrease** the program execution speed.

2) Function call indirections do not handle the increase in deltaxsize due to movement of the **global data variables**.
Hermes : Introduction

In this paper are as follows:

1) Hermes avoids the latency in the user program due to the use of indirection table.
2) Hermes eliminates the effect of global variable shifts on the size of the delta script.
Hermes: overview

Delta generation steps

Application level modifications

New user application

Function call indirection

Global variable placement

Byte level comparison

Delta script

Delta distribution stage

Delta script downloaded by nodes

Old application

Image rebuild and load stage

New application

Executed on host computer

Executed on sensor nodes
Hermes: Global variables

How different variables are stored in RAM. Initialized global variables are stored as `.data variables'. Uninitialized global variables are stored as `.bss variables'.
Hermes: Global variables

Hermes adds one stage (Structure generator) to the executable building process.

initialized global variables: `iglobStruct`
uninitialized global variables: `uglobStruct`.

Delta size of Case II: 280 bytes => 156 bytes
Hermes: DELTA DISTRIBUTION STAGE

- Reprogramming component
- Delta script
- User applications

Image 0
Image 1
Image 2
Image 3

Reprogramming component
Delta script
User applications

Image 0
Image 1
Image 2 and Image 3
Hermes: DELTA DISTRIBUTION STAGE

1) **image2** be the current version (v1).

2) Generate delta script

3) The **base node** broadcasts the command to reboot all nodes in the network from **image 0**.
### Hermes: DELTA DISTRIBUTION STAGE

<table>
<thead>
<tr>
<th>Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image 0</td>
</tr>
<tr>
<td>Image 1</td>
</tr>
<tr>
<td>Image 2</td>
</tr>
<tr>
<td>Image 3</td>
</tr>
</tbody>
</table>

6) All nodes receive the delta script and store it as **image 1**.

5) The user then injects the **delta script to the base node**.
The new image is stored as image 3.

In the next of reprogramming, image 3 becomes the old image and the is stored as image 2.
Avoiding latency due to indirection table:

When the bootloader reads call loc1, it finds from the indirection table that the actual target address for this call instruction is fun1.

So when writing to program memory, it writes call fun1 instead of call loc1.
Experiments

we considered a real world sensor network application deployed in Ross Ade football stadium at Purdue.
Experiments

Software change scenarios for TinyOS applications.

**Case 1**: Blink with a global variable added.

**Case 2**: Blink to CntToLeds.

**Case 3**: Blink to CntToLedsAndRfm.

**Case 4**: CntToLeds to CntToLedsAndRfm.

**Case A**: An application that samples **battery voltage** and **temperature** from MTS310 sensor where few functions are added.

**Case B**: We decided to use **opaque boxes** for the sensor nodes.
So, **few functions were deleted** to remove the light sampling functions.

**Case C**: In addition to temperature and battery, we added the features for sampling all the sensors on the MTS310 board except light (e.g. microphone, accelerometer, magnetometer).

**Case D**: Same as case C but with the addition of a feature to **reduce the frequency** of sampling battery voltage.

**Case E**: Same as case D but with the addition of a feature to filter out microphone samples (considering them as noise) if they are greater than some threshold value.
### Ratio of Number of Bytes to be Transmitted by Other Approaches to Hermes

<table>
<thead>
<tr>
<th></th>
<th>Deluge:Hermes</th>
<th>Stream:Hermes</th>
<th>Rsync:Hermes</th>
<th>Zephyr:Hermes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>148.62</td>
<td>84.92</td>
<td>63.47</td>
<td>39.04</td>
</tr>
<tr>
<td>Case 2</td>
<td>34.81</td>
<td>19.89</td>
<td>12.49</td>
<td>4.11</td>
</tr>
<tr>
<td>Case 3</td>
<td>12.37</td>
<td>7.66</td>
<td>5.64</td>
<td>2.73</td>
</tr>
<tr>
<td>Case 4</td>
<td>13.41</td>
<td>8.30</td>
<td>6.14</td>
<td>2.95</td>
</tr>
<tr>
<td>Case A</td>
<td>13.52</td>
<td>9.01</td>
<td>5.96</td>
<td>1.79</td>
</tr>
<tr>
<td>Case B</td>
<td>15.21</td>
<td>10.14</td>
<td>6.62</td>
<td>1.96</td>
</tr>
<tr>
<td>Case C</td>
<td>5.5</td>
<td>3.8</td>
<td>3.14</td>
<td>2.08</td>
</tr>
<tr>
<td>Case D</td>
<td>45.65</td>
<td>30.43</td>
<td>26.02</td>
<td>15.51</td>
</tr>
<tr>
<td>Case E</td>
<td>201.41</td>
<td>134.27</td>
<td>64.75</td>
<td>62.09</td>
</tr>
</tbody>
</table>
### TABLE II
**Ratio of Reprogramming Times of Other Approaches to Hermes**

<table>
<thead>
<tr>
<th></th>
<th>Deluge:Hermes</th>
<th>Stream:Hermes</th>
<th>Rsync:Hermes</th>
<th>Zephyr:Hermes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>24.77</td>
<td>44.66</td>
<td>34.24</td>
<td>14.12</td>
</tr>
<tr>
<td>Case 2</td>
<td>19.02</td>
<td>50.67</td>
<td>30.16</td>
<td>10.62</td>
</tr>
<tr>
<td>Case 3</td>
<td>6.14</td>
<td>13.48</td>
<td>9.80</td>
<td>4.77</td>
</tr>
<tr>
<td>Case 4</td>
<td>6.13</td>
<td>13.55</td>
<td>10.37</td>
<td>4.78</td>
</tr>
<tr>
<td>Case A</td>
<td>6.58</td>
<td>14.95</td>
<td>11.36</td>
<td>4.98</td>
</tr>
<tr>
<td>Case B</td>
<td>7.07</td>
<td>15.39</td>
<td>11.95</td>
<td>5.35</td>
</tr>
<tr>
<td>Case C</td>
<td>3.95</td>
<td>6.20</td>
<td>4.92</td>
<td>2.69</td>
</tr>
<tr>
<td>Case D</td>
<td>26.83</td>
<td>76.61</td>
<td>45.21</td>
<td>18.09</td>
</tr>
<tr>
<td>Case E</td>
<td>36.97</td>
<td>78.16</td>
<td>59.23</td>
<td>23.90</td>
</tr>
</tbody>
</table>

### TABLE III
**Ratio of Number of Packets Transmitted during Reprogramming by Other Approaches to Hermes**

<table>
<thead>
<tr>
<th></th>
<th>Deluge:Hermes</th>
<th>Stream:Hermes</th>
<th>Rsync:Hermes</th>
<th>Zephyr:Hermes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>28.54</td>
<td>140.31</td>
<td>91.83</td>
<td>17.05</td>
</tr>
<tr>
<td>Case 2</td>
<td>13.84</td>
<td>60.72</td>
<td>31.73</td>
<td>8.42</td>
</tr>
<tr>
<td>Case 3</td>
<td>5.93</td>
<td>13.03</td>
<td>10.40</td>
<td>4.16</td>
</tr>
<tr>
<td>Case 4</td>
<td>6.20</td>
<td>13.26</td>
<td>10.11</td>
<td>4.04</td>
</tr>
<tr>
<td>Case A</td>
<td>6.34</td>
<td>14.79</td>
<td>11.56</td>
<td>4.51</td>
</tr>
<tr>
<td>Case B</td>
<td>6.37</td>
<td>16.53</td>
<td>12.41</td>
<td>4.53</td>
</tr>
<tr>
<td>Case C</td>
<td>3.94</td>
<td>7.60</td>
<td>6.17</td>
<td>2.84</td>
</tr>
<tr>
<td>Case D</td>
<td>18.87</td>
<td>103.12</td>
<td>46.34</td>
<td>12.64</td>
</tr>
<tr>
<td>Case E</td>
<td>46.67</td>
<td>194.19</td>
<td>124.29</td>
<td>26.00</td>
</tr>
</tbody>
</table>
Hermes that minimizes the reprogramming overhead by reducing the size of the delta script that needs to be disseminated through the network.

Use techniques to mitigate the effects of global variable shifts and avoid the latency caused by function call indirections.