Efficient Code Distribution in Wireless Sensor Networks

Niels Reijers [*]
N.Reijers@its.tudelft.nl

Koen Langendoen
K.G.Langendoen@its.tudelft.nl

Faculty of Information Technology and Systems
Delft University of Technology
The Netherlands

ABSTRACT

The need to reprogramme a wireless sensor network may arise from changing application requirements, bug fixes, or during the application development cycle. Once deployed, it will be impractical at best to reach each individual node. Thus, a scheme is required to wirelessly reprogramme the nodes. We present an energy-efficient code distribution scheme to wirelessly update the code running in a sensor network. Energy is saved by distributing only the changes to the currently running code. The new code image is built using an edit script of commands that are easy to process by the nodes. A small change to the programme code can cause many changes to the binary code because the addresses of functions and data change. A naive approach to building the edit script string would result in a large script. We describe a number of optimisations and present experimental results showing that these significantly reduce the edit script size.

Categories and Subject Descriptors

C.2.2 [**Network Protocols**]: Applications; C.2.3 [**Network Operations**]: Network Management—*Reprogramming*; E.4.1 [**Coding and Information Theory**]: Data Compaction and Compression—*String distance*

General Terms

Design, Performance

Keywords

wireless, sensor networks, code distribution, reprogramming, compression, string distance

1. INTRODUCTION

Wireless sensor networks hold the promise for a wide range of new applications. These include intrusion detection, wildlife habitat monitoring [7], disaster management, and many military applications. The underlying technology that drives the emergence of sensor applications is the rapid development in the integration of digital circuitry, which will bring us small, cheap, autonomous sensor nodes in the near future. The capabilities of individual nodes are very limited, and since they are battery operated, energy conservation is a major concern.

[*]Supported by NWO (Dutch National Science Foundation) in the CONSENSUS project.

1.1 Motivation

There are many reasons nodes occasionally have to be reprogrammed. Application requirements change, and when nodes become cheap enough, it is possible that generic nodes will be integrated into roads, buildings, and other structures. These nodes can then be reprogrammed for a specific task when the need arises. Bug fixes and code updates are common to any software, and in the application development cycle one goes through a number of design-implement-test iterations.

It is clear that it is impractical at best to physically reach all nodes in a network, so a wireless reprogramming scheme is required. In this paper we present such a scheme, aimed at the requirements and restrictions specific to sensor networks.

Requirements

The limited capabilities of sensor nodes make cross-layer optimisations necessary. We expect the operating system and application to be more integrated than usual. These optimisations are necessary for power conservation and to squeeze the best performance out of very limited hardware.

This leads us to the first of our list of requirements:

1. The code distribution scheme should be able to update all code on the sensor node, including itself and the operating system. Also, it should not restrict the size of the programmes that can run on the nodes significantly, compared to when they are manually uploaded.
2. The scheme should be resilient to losing some packets during the process since nodes may operate in noisy conditions, have very simple radios, and cannot afford expensive transmission schemes.
3. Since communication is expensive in terms of energy, the code distribution scheme must limit communication as much as possible. Often changes to the code will be small. In those cases little communication should be required to distribute the new code.
4. Resources on the sensor nodes are scarce. The part of the distribution scheme running on the nodes should not require excessive processing, and should use little memory.
5. Finally, when updating code, we want the application to be stopped for only a short period of time.

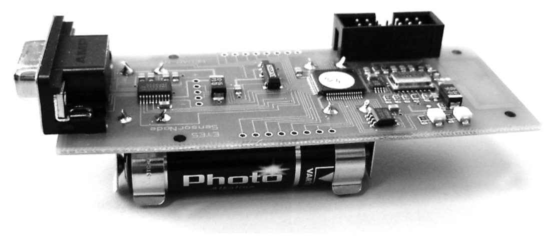

Figure 1: An EYES sensor node.

1.2 Related work

The need to reprogramme (wired) network nodes has also surfaced in other domains, in particular, in distributed systems ('mobile code') and active networks. The implementations of mobile code are based on virtual machines to guarantee some form of security and robustness [2, 11]. In the case of active networks people have experimented with virtual machines [8] as well as with native code [9]. None of the systems, however, supports modifying (parts of) the operating system, which is one of our main requirements.

We are aware of three systems in sensor networks that allow new applications to be loaded onto the nodes. In all three cases, code can be loaded onto a node, and this code then replicates itself onto other nodes by calling functions provided by the system.

SensorWare [1] allows users to programme the nodes using a scripting language. It provides a rich run-time environment. The scripting language is based on Tcl, extended with a number of commands to access the SensorWare functions. SensorWare is targeted at nodes that are much more powerful than ours.

A system aimed at very resource constrained sensor nodes was introduced as 'Maté' [6], and was later renamed 'Bombilla' and included in Tiny-OS. Bombilla is a stack-based virtual machine (VM), which provides a safe execution environment. To control the memory footprint of the VM, the programmer is limited to eight functions, called 'capsules', of at most 24 instructions. Although the instructions are quite powerful, this severely limits the sort of applications that can be built [1]. For instance, we were unable to implement the basic DV-Hop/Min-max localisation algorithm [5]. The algorithm does not fit into the eight functions Bombilla allows and the amount of variables is barely sufficient to hold the list of nodes with known locations.

The Berkeley Tiny-OS motes, models 'mica1' and below, for which Bombilla was developed require such an approach because the nodes cannot write to their own programme memory, but only to RAM. The scheme presented in this paper was designed for the EYES nodes, which do have the capability to update their Flash memory (code segment). This allows us to use native code instead of a VM making our scheme more flexible (i.e. updating OS code) and saves energy (i.e. interpretation overhead).

The Pushpin Computing System [4] is also aimed at resource constrained nodes. The Pushpin nodes can write to their own Flash memory. The basic blocks of application code are called a 'fragments'. Fragments contain native code and are restricted to 2KB of code and 445 bytes of state. Nodes communicate using a bulletin board system.

While all three systems allow new applications to be loaded, they cannot replace the underlying operating system.

1.3 The EYES nodes

The EYES nodes we used to implement our system on, shown in Figure 1, have a Texas Instruments MSP430F149 processor with 2KB RAM and 60KB of Flash memory to store code. The Flash memory can be programmed by the processor itself, or externally using a JTAG interface. The processor runs at a variable clock rate, with a maximum of 5MHz. Nodes communicate using a 115kbps radio (RFM TR1001 868.35 hybrid transceiver), and are equipped with an EEPROM memory module, which consists of four sectors of 64KB for a total of 256KB. The sectors can be erased independently. The Flash memory consist of 512 byte segments, that can also be erased independently. Figure 2 shows the layout of the address space on the EYES nodes.

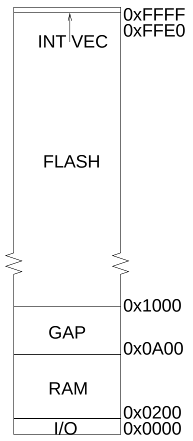

Figure 2: Layout of the address space layout of the EYES nodes.

We use an operating system that includes support for writing to Flash and EEPROM memory, as well as an energy efficient MAC protocol designed specifically for sensor networks [12]. The maximum message payload is set to 64 bytes.

2. CODE DISTRIBUTION

The envisioned procedure for code distribution consists of four stages (our current implementation is somewhat simpler):

1. Initialisation
2. Code image building
3. Verification
4. Loading

The procedure starts with an initialisation phase, informing the nodes that a reprogramming cycle has begun. Nodes make preparations like clearing the area where the new code should be build. In the second phase, a series of messages are sent to actually build the new code image. Once this is done, a verification step is used to check that each node has built the image correctly. Recovery from errors, like missed phase-2 packets, is done here. Finally, nodes are given the command to load the image and start running the updated code.

Of course, these four phases can be combined. Multiple phases can be dealt with in a single network packet.

2.1 Updating the running code

The requirement that we should be able to update all code on the node presents us with the problem that we need to overwrite the code that is currently running. We describe three options to do this. Which one is most appropriate depends on the available hardware.

Halve memory

For some applications, using halve of the available memory may be sufficient. In this case we can split the memory in an upper and lower halve, and, with the current code running in the upper halve, build the new code image in the lower one. Once the image is built, we place a small piece of code in RAM, and execute this to copy the bottom halve to the top halve and to reboot the node on completion. Since the processor is running code from RAM, there is no problem in overwriting the old code.

2-phase approach

The previous approach can be extended to use all available memory. We split the code into two halves, and place all the critical code required for our code distribution scheme in the bottom halve. The rest of the bottom, and the whole top halve can be used for application code. In the initialisation phase, we must now stop the application and clear the top memory halve. The critical code in the bottom halve keeps running. We then use the first approach to update the bottom halve containing the critical code. Once this is done, the new critical code is used to build the new top halve containing the rest of the application code. Only when this is done can we restart the application.

This approach has a number of drawbacks. Firstly, the application will be stopped during the whole process, instead of just during the copying of the new image. Secondly, we need to do the verification step for both halves, increasing the overhead of this approach. Finally, this approach is only possible if the critical code is smaller than halve of the memory, which should not be a problem in most cases.

Build in EEPROM

In our implementation we use the external EEPROM memory. We simply build the new code image in one of the EEPROM sectors, and use a small piece of code in RAM, as in the first option, to load the image into Flash memory. An added advantage of this is that once built, we can have multiple images stored in EEPROM, and load them when necessary.

2.2 Missed packet recovery

Since wireless sensor nodes may operate in noisy conditions, have simple radios, and do not have the resources for expensive communication protocols, it is likely that some nodes will miss one or more of the packets needed to build the new code image. Recovery from missed packets in the other phases is relatively straightforward since they contain less information and can be resent by a neighbour.

Each phase-2 packet contains all information needed to build a number of bytes of the new code image. Note that this area may be much longer than the length of the packet, because of the diff-like techniques described in sections 3 and 4. The packets are processed sequentially, building the new code image bottom up.

In Figure 3, we see that when a node receives packet n, it builds the code that packet n describes at the point where packet $n-1$ stopped. Instead of a sequence number to detect missed packets, we include the address of the first byte to be built in each packet.

For packet n, this is address x, which matches the address where the node stopped building after receiving the previous packet, $n-1$. But when the node receives packet m, it will find address z, while it expects address y. The node now knows that it has missed the packets that describe the area between y and z, and can record this information for later use. Because m contains all information needed to build the code at address z, the node can just continue building code from z.

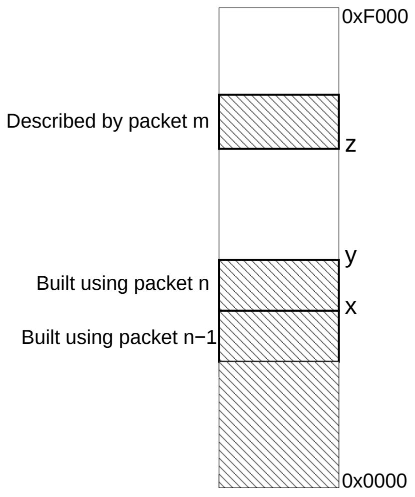

Figure 3: Missing a phase-2 packet.

If the missed packets are received at a later stage, the gap can be filled. If not, the node can ask its neighbours for the missing pieces of code during the verification phase. A neighbour node then sends a binary copy of the requested data, because the corresponding phase-2 packet will have been discarded long ago.

Since a node can process a packet immediately after receiving it and discard the buffer, the amount of state we need to store in RAM is minimised. This complies with our fourth requirement.

3. TRAFFIC REDUCTION

The third requirement was to be energy efficient by reducing the necessary communication. Since communication is very expensive compared to processing, we can afford to do some processing on the nodes if this reduces communication. At the same time, the amount of RAM is very limited. This rules out many decompression algorithms as they require too much RAM to store state.

3.1 A diff-like approach

Our code distribution scheme is based on the assumption that in many cases changes to the binary code will be small. Even when a whole new application needs to be loaded, it is likely that parts of the operating system and library functions will be present in both code images. This leads us to a solution similar to the `diff` command present in UNIX-like operating systems.

The UNIX `diff` system is based on finding the shortest string edit script from a source string to a destination string [10]. The operations that are allowed are insertion, deletion, and substitution of characters. Our scheme also finds such a shortest edit script, but using different operations.

Because of the limited commands in UNIX `diff`, the edit script from a string `aaaaabbbbbb` to `bbbbbbaaaaa`, is rather long. The `a`'s have to be deleted from the first string, and inserted after the `b`'s. Our command set contains two basic commands: `insert` and `copy`, leaving the deletes implicit. Using these commands, the string edit script would be:

```
copy 6 bytes from position 5
copy 5 bytes from position 0
```

The motivation for this different approach is that we expect pieces of code to move around from one version to another, and generic

Command	Opcode	Parameters	Cost (in bytes)
`insert`	`110xxxxx`	`xxxxx`:`len`, `data`	$1 + len$
	`11110000`	byte:`len`, `data`	$2 + len$
	`11110001`	word:`len`, `data`	$3 + len$
`copy`	`0xxxxxxx`	`xxxxxxx`:`len`, word:`from_addr`	3
	`11110010`	byte:`len`, word:`from_addr`	4
	`11110011`	word:`len`, word:`from_addr`	5
`repair`	`10xxxxxx`	`xxxxxx`:`offset`, word:`value`	3
	`11110100`	byte:`offset`, word:`value`	4
	`11110101`	word:`offset`, word:`value`	5
`repair dbl`	`11110110`	byte:`offset`, long:`value`	6
	`11110111`	word:`offset`, long:`value`	7
`patch list`	`11111000`	byte:`count`, {word:`from_addr` word:`to_addr` word:`shift by`}*	$2 + 6len$

Table 1: Edit script opcodes.

pieces of code to appear in both source and destination image (for instance pushing and popping certain registers at the start of a function). Being able to copy these parts of code, regardless of the order in which they appear, will reduce the edit script length, and thus reduce the communication cost.

We also expect that some pieces in the source and destination will be similar, but not identical (for instance a different operand in some generic sequence of instructions). We introduce a third command: `repair`, that operates on a `copy` command by replacing one or two words at a certain offset within the copied data. Using this will be shorter than a `copy`, `insert`, `copy` sequence.

3.1.1 Opcodes

We designed a set of binary opcodes to describe the edit scripts as efficiently as possible. The full set is shown in Table 1. The `patch list` command will be explained in Section 4.3. Different opcodes are available for most commands. This allows us to use a shortened version for small values of the operands. To increase the number of cases in which the operand value is low enough for a shortened version, we specify the length and offset operands in words, and store the actual length minus one. For instance, if the length of the data in an `insert` command is 64 bytes, the value stored in the length operand will be $\frac{64}{2} - 1 = 31$, which is the maximum value we can encode in the opcode. Since our maximum message payload is 64 bytes, we will in practice always be able to use this one-byte version of the `insert` command.

3.2 Edit script generation

Finding the shortest edit script for our command set is a non-trivial task. It is complicated by the facts that we have to consider copying non-perfect matches, and that the length, or cost, of a command depends on the value of its operands.

At the moment we use an optimal algorithm somewhat similar to the algorithm used in UNIX `diff`. Let the destination string D be $\{d_0, d_1, ..., d_n\}$, and D_m be the prefix $\{d_0, d_1, ..., d_m\}$ of D, with $m \leq n$. The algorithm starts by determining the cheapest way to build all prefixes of D, using either a single `insert`, or a single `copy` together with the required `repair`s.

This gives us the cheapest way of building D_0, and an initial upper bound on the cost for all other prefixes of D. Using this value of D_0, we then repeat the process to determine the cheapest way of building D_1, and try to improve the remaining upper bounds.

We check for all D_k, with $k \geq 1$, whether the current upper bound is higher than the cost of building D_0, plus the cost of building the rest of D_k using a single `insert` or a `copy` with the required `repair`s. If that is the case, we have found a better upper bound for D_k. For D_1, this is also the final cost, since there is no other way to build it and improve on its upper bound. We now know the cheapest way to build both D_0 and D_1.

We repeat this process until we find the lowest possible cost to build the whole of D, as well as the corresponding edit script.

We use a suffix tree data structure [3], which is a tree of all suffixes of the source string, to search for matching substrings. Because we have to look for non-perfect matches as well, we have to follow multiple parallel paths down the tree. A string with length n produces a tree that is $O(n)$ deep. The worst case number of parallel paths we have to follow is $O(n)$. Since we perform a traversal of the tree for each position of the string, the total complexity is $O(n^3)$.

On a 266MHz Pentium II it takes about four minutes to generate the edit script for a difficult case using binary images of about 20KB. An easy case of similar size takes roughly forty seconds. We expect this can be improved significantly by optimising the algorithm.

Splitting into packets

After the edit script has been generated, we need to split it into 64-byte packets. This causes some overhead because it should be possible to process each packet independently. We simply split `insert` commands when they do not fit in a packet. This costs one extra byte per split. We also need to split `copy` commands when a `copy` together with all its `repair`s does not fit. In that case we split the `copy` into two or more `copy` commands, and store each in a packet together with all the `repair`s that apply to the corresponding piece of copied data.

4. FURTHER OPTIMISATION

The scheme described so far already significantly reduces traffic compared to simply distributing the whole binary, but the script size is still larger than expected.

4.1 Address shifts

The large script size is caused by the fact that when we make (small) changes to the source code, the addresses of functions and data may shift. The compiled code from which the edit script is generated will differ in many positions because each reference to the shifted code or data now addresses a different memory location. This is illustrated in Figure 4, where a group of functions has moved from address `k` to address `m` because new code was inserted. This means that each call to `f` will be a `call` instruction to address

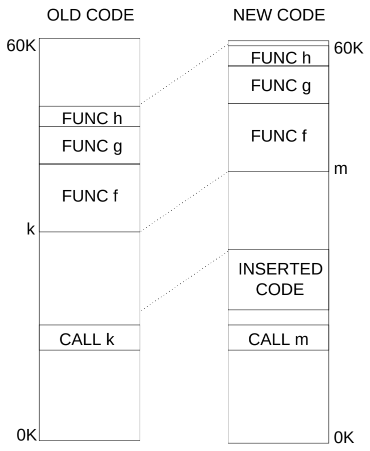

Figure 4: Address shifts due to inserted code.

m instead of address k in the new binary, and the corresponding instruction will have to be repaired in the edit script.

This problem is not limited to functions. Objects that may shift are functions and constant data, which are located in Flash, and global or local static variables, located in RAM. To assess the extent of the problem, we examined how many instructions in our operating system can be affected by address shifts of these objects. The results are shown in Table 2.

While function calls form the largest group, there are also many instructions that operate on an address in RAM. Only 291 out of 422 of these are `mov` instructions. The rest is made up of a variety of different instructions (`add`, `bic`, etc.). Further, we found many `mov`s of constant values to registers, and some `push` instructions, pushing constant values onto the stack. As Table 2 shows, these are mostly addresses. This means they too have to be repaired when the address changes.

In total, Table 2 shows us that 1073 out of 6707 instructions (16%) potentially have to be repairs when code or data they refer to, shifts.

4.2 Padding

The code on a node is often a collection of somewhat separate parts. Usually, changes will be limited to a few of these. A very simple way to reduce the address shift problem is to leave some empty space in between these parts, so that they can grow and shrink without moving the other parts. This can be applied both to data and code. Doing so limits the problem to references to the part that has changed, and no longer affects the rest of the code.

The use of this technique is limited. Once the space between parts runs out, we still have to move the part that is in the way, and once space for padding runs out all together, this technique can no longer be applied. The granularity at which the code is broken into parts, and the amount of padding space to leave between parts must be chosen wisely for this to work well.

Although padding does not solve the problem sufficiently, it may be a useful addition to the address patching technique described below.

4.3 Address patching

We can reduce the number of repairs due to address shifts by patching the address in, for instance, `CALL` instructions automatically. Since changes to the code are likely to be localised, groups of functions or data are moved together by the same offset (see

Instruction type	Count	Percentage
`call` instructions	475	7.1%
RAM access	422	6.3%
`mov` instructions	291	4.3%
RAM as source	141	2.1%
RAM to register	99	1.5%
RAM as destination	159	2.4%
register to RAM	65	0.96%
`mov` constant to register	193	2.9%
constant value is an address	158	2.4%
`push` constant onto stack	18	0.27%
constant value is an address	18	0.27%
Total number of instructions	6707	100%
Vulnerable to address shifts	1073	16%

Table 2: Number of instructions affected by address shifts in our operating system application (20KB code).

`f`, `g`, and `h` in Figure 4). Also, typically, a function is called more than once (if not, it would be in-lined by the compiler), and data addresses are used more than once. This means that many instructions are affected by the same shift. If nodes know about these shifts, they can patch the address of many of these instructions automatically when executing a `copy` command. We described the shifts by a list of {*begin address*, *end address*, *offset*}-tuples, and send these to the nodes using the `patch list` command shown in Table 1. For simplicity we want the entire patch list to fit into a single packet, limiting us to ten patches.

The format of the MSP430 instructions makes it easy to recognise instructions. Each instruction has a one word opcode, optionally followed by one or two words for operands. Now when executing a `copy` command, a node checks for each word that it copies if the previous word is the opcode of a patchable instruction. If so, and if the copied word, which is the used address, lies in the range covered by one of the entries in the patch list, we add the corresponding offset to the word value, and the instruction will use the right address.

The instructions we have considered for patching are:

1. function calls
2. loading a constant word value into a register
3. pushing a constant word value onto the stack
4. loading a word from memory into a register
5. storing a word from a register into memory

This set was determined during testing by selecting instructions that caused many `repair` commands in the edit script, and were suitable for patching. The second and third are commonly used to load the address of code or (constant) data.

Edit script generation with patching

When we use the address patching optimisation, we need to anticipate the patches the node will do when we generate the edit script. Therefore, we first apply the list of patches to the entire source image, in the same way as the nodes do. If the patching works well, this should reduce the number of differences between the source and destination image. Next, we run the edit script generation algorithm using the patched source and the normal destination image. This is shown in Figure 5.

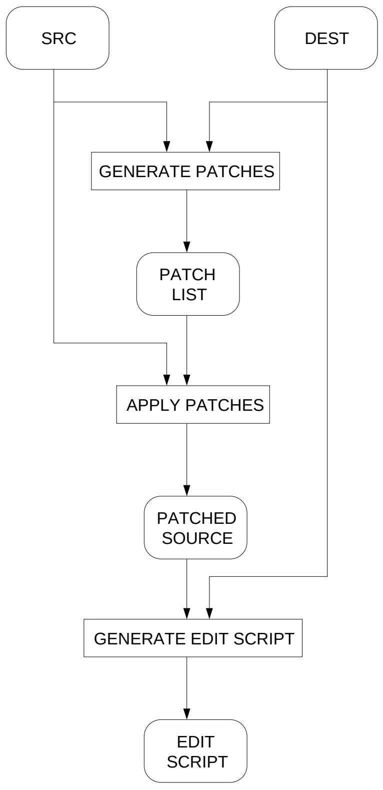

Figure 5: Patch list and edit script generation.

We do not check whether or not the recognised word is actually an opcode. It could also be a piece of constant data, or an operand of another instruction. Also, when we recognise an instruction that moves a constant into a register, the constant may not be an address at all. In these cases, nodes will incorrectly patch the following word if it matches an entry in the patch list.

These 'mispatches' cause more, instead of less, differences between the source and destination image. Since the edit script is generated using the patched source, it will contain `repair`s to correct them. While mispatches cause some extra bytes to be used, they do not affect the correctness of the generated script.

4.4 Patch list generation

The generation procedure for the patch list is simple. We use gcc to build an ELF executable from which we read the symbol table to determine the addresses of all `FUNC` and `OBJECT` type symbols, as well as the addresses of library functions. Once we have this list for both source and destination, we sort the source list on address, and try to find contiguous spans of symbols that appear in both lists and are shifted by the same amount. Each of these spans becomes a patch list entry. We then examine the disassembled code, and count for each patch list entry the number of instructions it applies to. Finally, we select the ten patches (if that many exist) that cover the most instruction. We also require a patch to cover at least two instructions, because patches that affect only one instruction can be handled more efficiently by a single `repair`.

5. EXPERIMENTAL RESULTS

We did several experiments to evaluate the performance of our scheme. The measure by with we compare the different optimisations is the length of the generated edit script, before splitting it into packets. Since the overhead incurred when splitting the script into packets depends on the packet length, we decided not to include this in the evaluation of our edit script algorithms.

Note that with our scheme, we can always use `insert` commands to build the entire image. In that case the overhead per packets is two bytes for storing the start address of the packet, and one byte for the `insert` opcode. At a packet length of 64 bytes, this is less than 5 percent overhead, which is the worst case scenario for our scheme.

For our experiments we used four different scenarios. The base code image is the latest version of our operating system, which includes some test applications. The base source code has 5027 lines of code, resulting in a binary image of 20308 bytes, including some libraries. The operating system consists of four main parts: kernel, networking, I/O, and application containing 1112, 1411, 694, and 1810 lines of code respectively.

5.1 'Code shift' and 'Data shift' scenarios

In the first two scenarios we generate the edit script from the base image to a slightly altered version to test the effectiveness of the address patching. For the first scenario we insert a single `noop` instruction at the lowest possible address to shift all code up by two bytes. This also shifts constant data, which is in the same segment. For the second scenario we shift the `.data` segment up by two bytes. This shifts all global and static local variables up by two bytes.

Patch and repair optimisations

Table 3 shows the result of generating the edit script with only `insert` and `copy` commands, with the `repair` and address patching optimisations turned on separately, and the result with both turned on.

We see that both the `repair` command and address patching significantly reduce the edit script size. For example, both optimisations combined improve the initial result for 'code shift' by 92%, and for 'data shift' by 60%. Address patching is especially successful when code moves, and much less when data moves. The reason for this is that patching calls, constant to register moves, and push constant instructions covers most of the instructions that contain addresses of code or constant data. The number of instructions addressing global data is much larger, as was shown previously in Table 2.

In both scenarios, using `repair`s allows us to repair the instructions that were not patched more cheaply than using an `insert` and a `copy`. Thus the result of both options combined is significantly better than the result obtained when just using address patching.

Set of patched instructions

In Table 4, we compare the result of patching for each instruction individually, by patching only that instruction and comparing the script length to the length of the script without any patching. Repairs are used in all cases. The cost of the `patch list` command in the edit script is listed separately and excluded from the individual cases since it is only incurred once when multiple instructions are patched.

We see that in the 'code shift' scenario, we benefit from patching calls, constant to register moves and push constant instructions. The 'code shift' scenario benefits from patching push constant instructions because the constant is usually a function pointer. Patching instructions that move data from memory into a register or back does not help in the 'code shift' case because RAM addresses do not change. The cost for the patch list is eight bytes. This is the two

Scenario	Copy/Insert	Repairs	Patching	Repairs and Patching
Code shift	4117	2640	429	326
Data shift	2920	1886	1720	1154
Small change	1649	1107	754	540
Major upgrade	9618	7722	7608	6308

Table 3: Resulting script size in bytes when applying repairs and/or patches.

Scenario	no patching	calls	const to reg	push const	mem to reg reg to mem	all	cost of patch list
Code shift	2640	-1586	-598	-53	0	-2322	+8
Data shift	1886	0	-166	0	-571	-740	+8
Small change	1107	-177	-153	0	-233	-581	+14
Major upgrade	7722	-864	-308	0	-260	-1476	+62

Table 4: Effect on script size of patching different opcodes.

bytes for the `patch list` command, plus six for a single patch list entry.

In the ‘data shift’ scenario, we benefit mostly from patching instructions that move data from memory into a register or back, as expected. Nothing is gained by patching calls or pushes. Both scenarios benefit from patching constant to register moves, since these are often either addresses of constant data, located in between code in Flash memory, or addresses in RAM.

The reason that improvement reported in the final column, when all patching options are turned on, is not the exact sum of the previous columns is related to the fact that we a repair with a small offset is cheaper than a repair with a large offset. This causes the script generation algorithm to split large pieces of copied code into several `copy` commands so a small instruction format for the `repair`s can be used (the offset within the `copy` is smaller). When more patches are done and less `repair`s are necessary, the need for this `copy` splitting is reduced, and we gain that on top of being able to drop the individual `repair` commands.

5.2 ‘Small change’ and ‘Major upgrade’ scenarios

The next two scenarios represent more realistic use of our code distribution scheme. For the first, we made a small change to the base code by adding 17 lines to our application code, including a static local variable. The application code is at the top of our binary image, so most of the code does not shift. When using our scheme it is best to keep the code that changes most frequently at the top of the image to reduce address shifting. Doing this does not require much effort from the programmers side.

In the ‘major upgrade’ scenario we took an older version of our operating system, and generated the edit script from that to the current base code. The source code of the old version we used contained 4328 lines of code (101KB), compared to 5027 lines in the base code (115KB). The changes from the old to the base version include adding ports to the network layer, adding some new test applications, and adding the code for our code distribution scheme. As an indication: the UNIX diff of the source codes produces a diff file of 48KB. The binary image of the old version was 16KB, compared to 20KB for the new one.

Patch and repair optimisations

The results in Table 3 again show that patching and repairs help improve the result, but not by as much as in the previous two cases. This is because in the previous two cases, none of the code actually changed, and we would ideally be able to copy everything. In the ‘small change’ and ‘major upgrade’ scenarios new code has been added, which cannot be copied. This is particularly clear in the ‘major upgrade’ scenario. In this scenario the destination binary was 4KB larger than the source. So a resulting edit script of 6KB seems acceptable since this includes both the 4KB of new code, quite some changes to the old code, as well as the overhead for moving data and code.

The resulting script size in the ‘small change’ scenario is small compared to the previous two scenarios, considering the fact that in this scenario both code and data shift. Because the changes were made at a location at the top of the image, less code and data was affected by this, which explains the small script size.

Set of patched instructions

Looking at the results for the different patch options in Table 4, we basically see the results of the previous two scenarios combined. Again, the results of patching push constant instructions are poor. The other options all result in a shorter edit script.

5.3 Current implementation

The current implementation we use on our nodes is somewhat simpler than the planned final result we describe here. However, the most important parts of the scheme have been implemented. What remains are a number of improvements that need to be implemented in order to build the complete scheme described here.

The edit script generation algorithm has been completely implemented. The distribution of the edit script however, is less sophisticated than it should be. We have not yet implemented a flooding algorithm. Currently, we use node to node communication with acknowledgements to transfer the edit script. We use only two nodes where one reprogrammes the other.

As an indication, in experiments with real nodes we found that, with the script loaded into EEPROM, the time needed for a node to build and load a new image was in the order of a few seconds. The bottleneck in this test is mainly the time needed to write to EEPROM and Flash memory. The script interpretation overhead is low. Of course in a real application flooding the script messages, which are then interpreted on-the-fly instead of stored in EEPROM first, will dominate the update time.

6. CONCLUSIONS

We presented a scheme to distribute code in wireless sensor networks. The scheme is simple to implement, and requires few resources on the sensor nodes. Different options were presented that allow the scheme to be used on nodes with various hardware.

The scheme is resilient to missing packets in that it can continue processing the following packets and start a recovery procedure in a later phase.

Updating all software on the nodes is possible, including the operating system and the code distribution scheme itself. The scheme distributes binary native code, so the programmer is not bound to a virtual machine, but can do all low level optimisations necessary when programming for wireless sensor networks.

A diff-like algorithm and two optimisations were presented (repairing and patching) that reduce the necessary communication, especially when the currently running code is similar to the code that is to be distributed. Experimental results were presented, showing that these significantly reduce the amount of communication compared to simply transferring the binary code. Other experiments showed that the application only has to be stopped for a few seconds when distributing new code.

Most of our scheme has been implemented and has been used to update code on real sensor nodes.

Future work

There are many opportunities to further improve our scheme. The opcodes of the edit script commands can be tuned further to reduce script size. Also, more work needs to be done on the script generation algorithm. The speed of the algorithm needs to be improved, and it may be worth looking into good heuristics to replace the optimal algorithm. Also it would help to consider the packet boundaries when generating the edit script to reduce the overhead incurred when splitting the script. The address patching may be improved by expanding the set of patchable instructions. Finally, a flooding algorithm needs to be implemented to do code distribution to multiple nodes, and the recovery procedure for missed packets needs to be developed.

7. REFERENCES

[1] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, May 2003.

[2] A. Carzaniga, G. Picco, and G. Vigna. Designing distributed applications with a mobile code paradigm. In *ICSE*, pages 22–32, Boston, MA, May 1997.

[3] D. Gusfield. *Algorithms on Strings, Trees and Sequences*, chapter 6.1: Linear-Time Construction of Suffix Trees, Ukkonen's linear-time suffix tree algorithm. Cambridge University Press, June 1997.

[4] J. Lifton, D. Seetharam, M. Broxton, and J. Paradiso. Pushpin Computing System Overview: A Platform for Distributed, Embedded, Ubiquitous Sensor Networks. *Proceedings of the International Conference on Pervasive Computing*, August 2002.

[5] K. Langendoen and N. Reijers. Distributed localization in wireless sensor networks: A quantitative comparison. *Computer Networks, special issue on Wireless Sensor Networs*, August (accepted for publication), 2003.

[6] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *ACM ASPLOS*, pages 85–95, San Jose, CA, October 2002.

[7] 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)*, pages 88–97, Atlanta, GA, September 2002.

[8] J. Moore, M. Hicks, and S. Nettles. Practical programmable packets. In *IEEE INFOCOM*, pages 41–50, Anchorage, Alaska, April 2001.

[9] E. Nygren, S. Garland, and F. Kaashoek. PAN: A high-performance active network node supporting multiple mobile code systems. In *IEEE OpenArch*, pages 78–89, New York, NY, March 1999.

[10] G. A. Stephen. String searching algorithms. In D.T. Lee, editor, *Lectures Notes Series on Computing*, volume 3, chapter 3, String Distance and Common Sequences, pages 39–86. World Scientific, Singapore, 1994.

[11] P. Tullmann and J. Lepreau. Nested Java processes: OS structure for mobile code. In *ACM SIGOPS European Workshop*, pages 111–117, Sintra, Portugal, September 1998.

[12] T. van Dam and K. Langendoen. An adaptive energy-efficient MAC protocol for wireless sensor networks. In *1st ACM Conf. on Embedded Networked Sensor Systems (SenSys 2003)*, Los Angeles, CA, November (accepted for publication), 2003.