DYNAMIC SOFTWARE UPDATE OF RESOURCE-CONSTRAINED DISTRIBUTED EMBEDDED SYSTEMS

Meik Felser, Rüdiger Kapitza,
Jürgen Kleinöder, Wolfgang Schröder-Preikschat
Friedrich-Alexander University of Erlangen–Nuremberg
Dept. of Computer Sciences 4 (Distributed Systems and Operating Systems)
Martensstr. 1, 91058 Erlangen, Germany
E-Mail: { felser, kapitza, jk, wosch } @cs.fau.de

Abstract Changing demands, software evolution, and bug fixes require the possibility to update applications as well as system software of embedded devices. Systems that perform updates of resource-constrained nodes are available, but most approaches require a complete restart of the node after installing or updating software. Restarting the node results in the loss of important system state, such as routing information or sensor calibration values. Rebuilding this information requires time and energy.

In this paper we present an online state-preserving update system for resource-constrained nodes. A remote incremental linking approach is used to generate node-specific and execution-state dependent code. Compiler-generated symbol, relocation, and debugging information is used to determine whether a dynamic update of the running system is possible and how it can be achieved.

Keywords: dynamic update, managed nodes, ELF, relocations

1. Introduction

The number of microcontrollers used in today's embedded systems is increasing. In more complex systems, a network of several different microcontrollers is used, thereby forming a heterogeneous system of larger and smaller nodes. Especially, we have to face distributed embedded systems with a mixture of very resource-constrained devices on the one hand and more powerful devices on the other hand.

An example of such a network is a sensor network [1], a large number of small, non-expensive, and very resource-constrained nodes equipped with sensors to collect environmental data and able to communicate. Usually the sensor nodes are supported and managed by a larger, more powerful node, the base station.

Once initially set up, selective software updates will become necessary due to bug fixes, improved functions, extensions, and parameter changes. For example, if the calibration process did not deliver a set of parameters that yields the desired results, software modifications become necessary. Replacing parts, such as the sampling or pre-processing algorithm, or adding additional software to use another sensor will be the consequence. Furthermore, with an estimated lifetime of several years, a future-proof system must be able to replace any part of the system such as the scheduling system or the communication protocol.

When updating the software of a node it is desirable to do this with as little impact as possible. Considering the limited resources of sensor nodes, the update process itself must not use a lot of resources. Furthermore, different updates affect different parts of the system and updates of the application should not affect system services. It is not acceptable that the addition of a sensor driver results in the restart of the complete node. When restarting the node, state information of applications and the operating system library is lost. Multihop communication protocols, for example, store routing information to represent the network topology. After a restart such information is lost and has to be rebuilt. The routing of the network has to be redetermined, which does not only affect the local node but the communication of other nodes, too.

Our goal is to update sensor nodes dynamically thereby preserving application and system state. During the update process the nodes are stopped but no restart is needed after the update is complete. The advantage is a considerable shorter off-line time and a better performance resulting from a very quick recovery after the update because the system state does not have to be rebuilt unnecessarily.

We describe an update infrastructure that allows dynamic reprogramming of nodes based on the binary code of the application. Most parts of our update system are located at a more powerful node, the base station for example. We use compiler-generated information, such as symbol tables, relocation tables, and debugging information, to identify situations in which a safe update is possible. In rare cases, the execution state might prevent a dynamic update. Such situations are signalled to the administrator so that additional actions can be taken, such as a redesign of the application to increase the updatability.

The following section gives an overview about related approaches for updating nodes in sensor networks. Section 3 outlines the architecture of our system. After that the object file analysis is described followed by section 5 presenting the procedure for updating or replacing code. Section 6 concludes the paper and gives a short overview of the current prototype.

2. Related Work

There are several approaches for dynamically updating software. In the following we will discuss some existing approaches for updating sensor nodes.

TinyOS [7] provides XNP [2, 10] to update a sensor node. This approach only supports very coarse-grained updates because the complete memory image is replaced. It has a huge impact if only some parameters are to be adjusted. Deluge [8] provides similar possibilities and uses Trickle [13] as multihop dissemination protocol to distribute the update to multiple sensor nodes. Both approaches need to restart the sensor node, because a new memory image is installed.

FlexCup [15] is the update system of TinyCubus [16], a framework to configure nodes in a role-based way. Components are transmitted as relocatable binary images including symbol tables and relocation information. A linker component on each node integrates the new components into one image. The system is restarted after the update to eliminate dangling pointers. Our approach does not need the linker component on each node because that part of the process is performed remotely.

Koshy and Pandey [11] present an approach similar to ours. They modify the development toolchain to create an incremental linker that is used to prepare the updated code on the base station. Thus, they can control where modified functions are to be placed on the node and reduce modifications resulting from moved code. They do not consider the current state of the system and perform a restart after the updated code is written into flash memory.

There are projects that support updating of a running system. To find all references to the code they often use an indirection layer to access the updatable code. One example is SOS [5], an operating system for sensor nodes that is built of modules. Modules can be updated, removed, and replaced at run time. The modules are transmitted and installed in a relocatable binary image format. Module code is position-independent by using only relative jumps, thus limiting the maximum size of a module to 4 Kbytes, the maximum distance of relative jumps on the target platform, the AVR ATmega128. Furthermore, references to functions or data outside of the module are implemented via an indirection table or are not allowed, respectively. Contiki [3] works similar and also provides a framework for dynamically loading libraries. The libraries contain relocation information that enables the installation of the code. Access to the libraries is provided indirectly via stubs. Our approach does not rely on an indirection table; we identify and modify the references directly. With this approach, we can even update code of the operating system library or the kernel.

3. Architecture

To achieve a minimal update infrastructure on the resource-constrained node, most parts of the update preparations are executed by a remote node. That node is called the *manager* and is usually more powerful and equipped with several megabytes of memory. The manager node is responsible for updating and managing the software installed on the resource-limited nodes (Fig. 1).

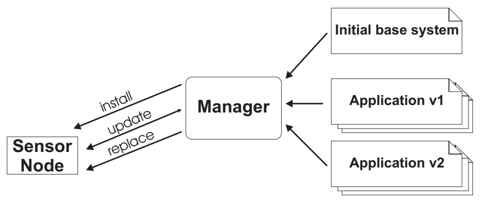


Figure 1. Duties and responsibilities of the *manager*.

As we will discuss in section 5 some situations preserve the automatic update of the running system. The information automatically extracted is not sufficient to decide whether the update of the running system can be performed. In these cases the administrator, who runs and supervises the system, is informed. He can the take the decision or use the information to redesign the application in a way that the manager can decide the situation automatically the next time.

Figure 2 shows a schematic overview of the manager node. The next paragraphs give an overview about the most important elements.

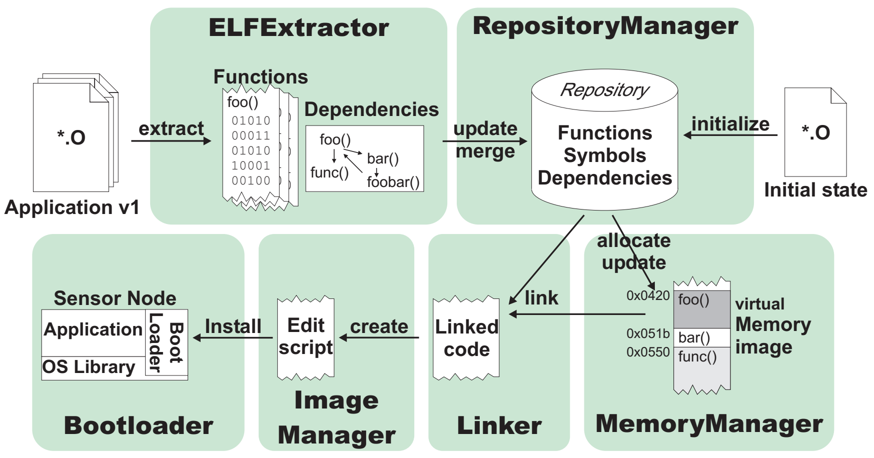


Figure 2. Architecture of the manager node.

3.1 Repository Manager

The *repository manager* is responsible for managing and analysing the code. The code is provided as binary object files. Software that is already installed and currently running on a sensor node is called the initial image of a node. The repository manager determines which functions, more precisely which symbols, are available in the compiled and linked ELF [22] object file of the initial image. Software that is to be updated or installed on a node has to be provided as *relocatable* ELF object files.

The *ELFExtractor* loads these binary files and analyzes the contents of each input file to identify individual functions and data objects. This results in a fine-grained modularization of the application. After that the repository manager extracts the dependencies between the identified objects from the relocation information and builds the dependency graph to determine which data is needed by a function and which other functions are called. More details about the modularization process are given in section 4.

When the administrator changes a function the system identifies which functions and data in the dependency graph are affected and where these parts are installed in the network. From this analysis a set of differences for each node can be calculated. This set of changed functions and their dependencies present the basis of semi-automatic policies that are used to determine whether the update operation can be performed. In case of conflicts the administrator gets involved to control the update operation. This is addressed in detail in section 5.

3.2 Image Manager and Memory Manager

The *image manager* creates and manages a memory image model that represents the usage of the memory in the device. At first this memory model is initialized with the initial state of the node. The model is then used for keeping track of the currently installed software layout. Based on this information the *memory manager*, a part of the cross linker, determines a location where to put modified code on the node. For example, an updated function should be located at the same position as the original function to minimize the number of references that need to be updated. The virtual image includes a list of application modules that are currently installed on the node including their exact position.

The image manager also records changes the linker performs to update the application. Based on this information it calculates a change set after the new functions were linked into the virtual image. This set contains the differences for each updated function and the code of all new functions. Deleted functions are not in the change set. The system ensures that these functions are not used anymore and the memory manager reuses their space to store new functions.

The changes can then be transmitted to the node in form of edit commands as they are generated by common diff algorithms, such as UNIX diff [9] or Xdelta [14].

3.3 Bootloader

A small bootloader resides on each node and is able to receive and process commands from the manager. Its main task is installing new code but it also provides the manager with essential information about the current state of the node. As described in section 5 the current state influences whether an update of the running system is possible or not.

When a sensor node receives an edit command, the application transfers control to the bootloader which either extracts the requested data or modifies the SRAM and flash memory of the sensor node.

The actual transfer of code blocks can be done via several protocols. Either direct single-hop communication or more complex multihop protocols like Trickle [13] or MOAP [20] are an option.

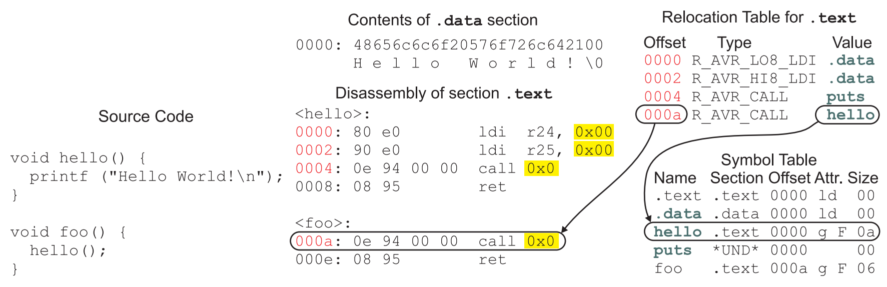
4. Modularisation and Dependency Analysis

Functions and data objects are identified by analysing the relocatable ELF object files. The analysis is solely based on the information provided in this ELF files. The developer does not need to use special constructs or annotations to support the analysis.

ELF files are organized in sections. Some sections contain symbol tables and relocation tables, other sections contain the relocatable binary data. A simple example is shown in figure 3. To determine the dependencies, we use the symbol and relocation information an ELF object file provides. For each relocation the position inside the associated binary section is given at which the final address of a symbol should be inserted. The final address of a symbol is resolved by the linker as soon as the location of the associated section inside the target's address space is determined [12].

To extract a function we look at the symbols associated with the `.text` section. These symbols give us the name, the start offset, and the size of the code. Thus we know where each function starts and how much space it allocates. To determine the dependencies of a function we take advantage of the relocation table. Each relocation that affects the code of the function represents a dependency.

We instruct the compiler to place each function in a separate section in the object file. This way we ensure that each function is visible and each dependency is represented by a relocation embedded in the ELF file. Otherwise calls of local (`static`) functions might not be visible in the relocation table. In the example in figure 3 both functions are in the same section. Thus the com-


Source Code
Contents of .data section
Disassembly of section .text
Relocation Table for .text
Symbol Table

```
void hello() {
  printf ("Hello World!\n");
}

void foo() {
  hello();
}
```

```
0000: 48656c6c6f20576f726c642100
      H e l l o   W o r l d ! \0
```

```
<hello>:
0000: 80 e0        ldi   r24, 0x00
0002: 90 e0        ldi   r25, 0x00
0004: 0e 94 00 00  call  0x0
0008: 08 95        ret

<foo>:
000a: 0e 94 00 00  call  0x0
000e: 08 95        ret
```

Offset	Type	Value
0000	R_AVR_LO8_LDI	.data
0002	R_AVR_HI8_LDI	.data
0004	R_AVR_CALL	puts
000a	R_AVR_CALL	hello

Name	Section	Offset	Attr.	Size
.text	.text	0000	ld	00
.data	.data	0000	ld	00
hello	.text	0000	g F	0a
puts	*UND*	0000		00
foo	.text	000a	g F	06

(a) source code (b) ELF file contents

Figure 3. Example of the data contained in an ELF file and its relationship

piler knows the distance between the two functions and the call from `foo` to `hello` could be done via a relative call without a relocation entry. If `hello` is additionally static, the compiler might even suppress the symbol.

We do not necessarily have to apply the same approach for the data objects of a program, because they are never placed in the same section as the functions. Thus, the symbols, even of local data, are available because the compiler needs them to insert references to data. Nevertheless, we instruct the compiler to generate separate sections because this makes it easier to identify each single data object. Otherwise the compiler creates one symbol for several local variables and just uses different addends and a platform-specific analysis would be necessary to separate such data objects.

5. Dynamic Code Update

Replacing or updating code in a running system requires knowledge about all references to the old code and their substitution with references to the new code. We also have to adjust the target addresses of jumps if a function is moved to another address. We further have to take special care if the function is modified while it is used by a thread of control. An overview of problems and possible solutions when updating software dynamically is given in [6]. Here we address three problem classes:

- references to a function as it occurs in function calls
- active functions that are currently executed by the target
- functions on the call stack (the function called a subfunction that is currently active)
- update of global variables

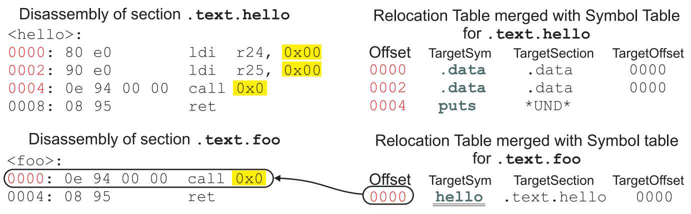
In the following subsections we examine and discuss these problems.

5.1 References to Functions

The first question we have to address is whether the updated function starts at the same address as the old one. If the new code has the same size or is smaller than the original function it fits into the old space and can start at the same address. This is the preferable situation because we do not need to change any references. To allow even larger updates fit into the space of a function the memory manager may allocate extra space [18] when initially placing a function. In resource-constrained environments this approach has to be applied carefully, because the additional space is wasted as long as it is not used by a replaced function.

If the modified function is larger and does not fit into the space of the old function it has to be installed at a different location. Then we have to identify all references to this function and update them.

References to the start of a function are used in conjunction with calls. Ordinary function calls can be detected by the use of symbol and relocation information. The relocation table for a function that calls another function contains an entry specifying the location where the address of the target function must be inserted (fig. 4). We use this information to find and update the reference to the function.



Disassembly of section **.text.hello**

```
<hello>:
0000: 80 e0          ldi  r24, 0x00
0002: 90 e0          ldi  r25, 0x00
0004: 0e 94 00 00    call 0x0
0008: 08 95          ret
```

Relocation Table merged with Symbol Table for **.text.hello**

Offset	TargetSym	TargetSection	TargetOffset
0000	**.data**	.data	0000
0002	**.data**	.data	0000
0004	**puts**	*UND*	

Disassembly of section **.text.foo**

```
<foo>:
0000: 0e 94 00 00    call 0x0
0004: 08 95          ret
```

Relocation Table merged with Symbol table for **.text.foo**

Offset	TargetSym	TargetSection	TargetOffset
0000	**hello**	.text.hello	0000

Figure 4. The relocation information shows that foo needs hello. If we update hello and thereby move it to an new position we need to modify foo as well.

.

Before patching the location with the new address, we have to determine how the address is used. It may either be used in a call operation or it may be stored in a variable to do an indirect call at a later time. We can easily patch the call operations with the new address. In all other cases it can not be exactly determined when and by which function the address will be used. In the worst case the address of the function is stored in a global variable for use by other functions. A combined code and data flow analysis could be able to detect this case but we do not think that the cost for this highly architecture-dependant operation is justified.

Our approach is to identify whether there is enough information to correctly find and patch all references. As a consequence we cannot patch calls via function pointers and, thus, forbid the use of function pointers, at least for functions that should be updated. Nevertheless, function pointers are allowed in well-known code, such as interrupt vector tables and the operating system scheduler. A system-specific layer is used to identify code and data references in this code.

This is not a strong limitation as function pointers are very rarely used in embedded applications[1]. Furthermore, function pointers are an additional source of errors especially for inexperienced programmers. For the same reason other programming paradigms for embedded systems, such as the Misra C rules [17], forbid the use of function pointers at all.

If our system detects the address of the updated function in any other than a call operation it indicates an insecure situation to the administrator who has to decide whether the function should be updated nevertheless.

5.2 Active Function

Before actually updating a function we have to make sure that it is not currently in use. The consequences of such an update are unpredictable. A function is in use if it is interrupted by the update process, as shown in figure 5, or if a thread was executing the function before it was suspended. In some event-driven systems, such as TinyOS [7], this situation does not occur as they do not offer a thread concept. Just event handlers can interrupt the normal control flow. Thus the update process gets scheduled as a task when no other task is active. Other systems, such as Contiki [3] or Nut/OS [4] offer a thread concept.

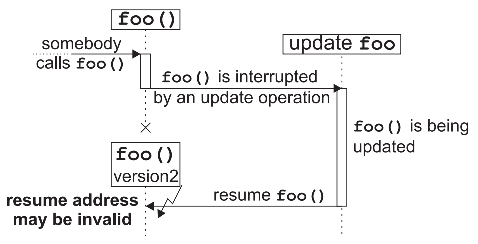


Figure 5. A function is interrupted and updated while it is active. Resuming the execution at the interrupted address may result in a crash or unpredictable behaviour.

To identify active functions, the manager asks the bootloader at which address the system would resume operation when exiting the loader. If the system supports multiple threads the bootloader returns the current position of all threads. The update manager checks whether the update would affect one of these addresses and indicates an update problem.

[1]An analysis of typical TinyOS applications showed that usually no function pointers are used. Nevertheless pointers to functions are used by the TinyOS scheduler and implicitly in the interrupt vector table.

A simple but promising approach to overcome this situation is to resume the system and retry the update a few moments later. If we are updating a sensor reading function it is likely that the function is not in use the next time we try an update. To avoid the periodic check, we can improve this approach by modifying the return address in a way that the bootloader is called as soon as the function returns. We then resume normal operation and wait until the function returns into the bootloader. If the function is still in use after a timeout or several retries, we inform the administrator about the failed update. Then he can authorize a reset after the update.

The success of this approach depends on the modularity of the application. Sensor net applications developed with some sort of framework, like TinyOS, are often designed very modular. Furthermore we inform the administrator about the reasons of a failed update. He can use this information to improve the application's modularity with regard to the next update.

5.3 Functions on the Call Stack

Beside functions that are directly in use by a thread, functions may also be on the call stack. That is, the function that is to be replaced called another function and this function is currently active. When the called function returns it uses the return address on the stack to resume operation in the caller. If we replace the code of the caller this return address might become invalid. Figure 6 shows an example. `foo` calls `hello`, this may be a call from a sensor calibration function to the function that actually gets the sensor readings. During the execution of `hello` the function `foo` is updated. `hello` is not influenced directly but the code of `foo` is modified and the return address that is used to resume `foo` after `hello` returns may be incorrect or invalid.

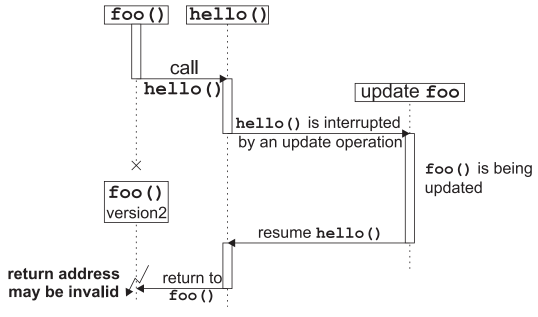


Figure 6. A function on the call stack is updated. When the flow of control returns to the updated function the resume address may be invalid.

We do not want to forbid the update of such functions in general, because minor changes, such as bug fixes, may alter the addresses but not the functionality of the code. Thus, updates are allowed if they do not alter the structure of the function. The new code must have the same calls to the same subfunctions as the old code and the same local variables. In our example above, we can fix a bug in the calibration algorithm `foo` as long as we still call the reading function `hello` and do not insert or delete local variables. We then assume that the return address A of the n-th call to a specific function from the old code can be replaced with the equivalent return address A' in the new code.

To accomplish this we have to know the stack layout and position. With this information we can walk through the stack and detect each function on the stack. The return addresses can then be adapted. The information how to obtain the stacks is encoded in the bootloader. The manager transfers a mapping from old to new return addresses if the loader requests it.

The debug information can give us information about the local variables of a function. Thus we can perform an offline check whether new local variables were added or removed.

If the structure of the function is modified we can retry the update several times. The function is possibly not in use anymore at a future retry. But the deeper the function is on the call stack the smaller is the probability that the function is not in use. If the retries are not successful, we inform the administrator about the failed update. Again, he can authorize a reset and use this information for the development of future versions of the application.

5.4 Update of Variables

Update of local variables is not supported as mentioned in the previous sections. Updates of global variables are allowed under certain conditions. If a function references a new global variable, this variable is simply allocated on the node. In the opposite case, that a variable is no longer used by an updated function we can not reliably determine whether the variable is used by some other code. Therefore we ask the administrator if the variable can be freed if we do not detect any references anymore. If a variable with the same name but different type is introduced we have several possibilities. If the old variable is still used by some code, we generate a warning because this situation most likely leads to inconsistencies and the administrator has to decide whether to continue with the update or not. If the new variable replaces the old the current state should be transformed. Without further information from the administrator this is not possible. An automatic transformation is subject to the same restrictions as, for example, the object (de-)serialization in Java [21].

6. Conclusions and Current Status

The presented work builds a cornerstone of our long-term objective to provide support for robust and efficient software management in heterogeneous sensor networks [19]. It enables the management of software installed on resource-constrained nodes by providing the ability to update and replace code in the running system. This is achieved by incrementally link new code to the existing application and to identify unused code. Necessary configuration information is stored at a larger node that prepares the update. This manager node also checks whether an update is possible in the running system at all. In critical situations, the administrator gets involved who can control and decide in which way the update has to be performed.

The current prototype of the manager software is implemented in Java. It can load and analyze relocatable ELF object files for x86, Hitachi H8 and AVR CPUs. Up to now, the prototype communicates and updates single nodes but it will be extended to handle groups of equal or similar nodes.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38(4):393–422, Mar. 2002.

[2] Crossbow Technology, Inc. *Mote In-Network Programming User Reference*, version 20030315 edition, 2003.

[3] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th IEEE Int. Conf. on Local Computer Networks (LCN'04)*, pages 455–462, Nov. 2004.

[4] egnite Software GmbH. *Ethernut Software Manuals*, Nov. 2005.

[5] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *3rd Int. Conf. on Mobile Systems, Applications, and Services (Mobisys '05)*, pages 163–176, June 2005.

[6] M. Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, Department of Computer and Information Science, Aug. 2001.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, pages 93–104, Nov. 2000.

[8] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys'04)*, pages 81–94, Nov. 2004.

[9] J. W. Hunt and M. D. McIlroy. An algorithm for differential file comparison. Technical Report 41, Bell Telephone Laboratories, 1976.

[10] J. Jeong, S. Kim, and A. Broad. Network reprogramming. Technical report, University of California at Berkeley, Aug. 2003.

[11] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *2nd Europ. W'shop on Wireless Sensor Networks (EWSN 2005)*, pages 354–365, 2005.

[12] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, San Francisco, CA, USA, Oct. 1999.

[13] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *1st Symp. on Networked System Design and Implementation (NSDI '04)*, pages 15–28, Mar. 2004.

[14] J. P. MacDonald. File system support for delta compression. Master's thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 2000.

[15] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *3rd Europ. W'shop on Wireless Sensor Networks (EWSN 2006)*, volume 3868 of *LNCS*, pages 212–227. Springer, Feb. 2006.

[16] P. J. Marrón, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A Flexible and Adaptive Framework for Sensor Networks. In *2nd Europ. W'shop on Wireless Sensor Networks (EWSN 2005)*, pages 278–289, Jan. 2005.

[17] MISRA. *MISRA-C: 2004 - Guidelines for the use of the C language in critical systems*, Oct. 14, 2004.

[18] R. W. Quong and M. A. Linton. Linking programs incrementally. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):1–20, Jan. 1991.

[19] W. Schröder-Preikschat, R. Kapitza, J. Kleinöder, M. Felser, K. Karmeier, T. H. Labella, and F. Dressler. Robust and efficient software management in sensor networks. In *2nd IEEE/ACM International Workshop on Software for Sensor Networks (SensorWare 2007)*, Jan. 2007.

[20] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS–TR–30, University of California, Los Angeles, Center for Embedded Networked Computing, Nov. 2003.

[21] Sun Microsystems. *Java Object Serialization Specification*, 2001.

[22] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, version 1.2 edition, May 1995.