Dynamic Software Management on BTnode Sensors

Falko Dressler, Moritz Strübe, Rüdiger Kapitza and Wolfgang Schröder-Preikschat
Dept. of Computer Science, University of Erlangen-Nuremberg, Germany
Email: {dressler,kapitza,wosch}@informatik.uni-erlangen.de

***Abstract*—Efficient software management is one of the great challenges in sensor networking. The main objective is to handle heterogeneity and dynamics. Based on recently developed architectures such as our profile-matching approach, decisions can be taken on a higher level about which software module to run on which sensor node. Going to the physical network, the idea is still to replace the complete binary image of the sensor node application. This process is highly resource expensive. We developed a system that allows to add, to remove, and to replace single modules on a running sensor node without restarting the node. Additionally, the entire kernel can be replaced. One of the most prominent features is the ability to keep the local state of the node, i.e. variables as well as dynamically allocated memory and to restore it after replacing the module.**

I. Introduction

Sensor networks still represent a growing and challenging research domain [1], [2]. With some first real-world application scenarios such as habitat monitoring [3] and precision agriculture [4], open issues can continuously be identified and challenged, and appropriate solutions can be evaluated. Many challenges, such as energy efficiency, security, and self-organization, have been identified in this area [5], including also software management in sensor networks [6], [7].

According to Han and co-authors [7], software management in sensor networks consists of three fundamental components: the execution environment at the sensor node, the software distribution protocol in the network, and the optimization of transmitted updates. We concentrate on software management techniques for sensor networks that are dynamic in terms of availability, mobility, and current application demands. In order to support heterogeneous hardware platforms and to address issues such as the low resources in terms of processing power, available memory, and networking capacities (sensor nodes are often able to run a single task only) [8], new approaches for efficient software engineering are needed. An overview to the issues that are specific for sensor nodes is provided by Culler et al. [9].

For on-demand software updates and general node reprogramming, two things are needed. First, an architecture that maintains an overview of the sensor network and all available resources in general. This architecture must be able to take decisions where to run which program in the network based on an internal decision process. Secondly, techniques are needed to actually upload software to the sensor nodes. We will outline some relevant approaches in the next paragraphs.

In general, two approaches for software updates in sensor networks have been discussed in the literature: multihop network-based node reprogramming and robot-assisted software management. Work on the first technique was done mainly based on network-centric reprogramming. For example, the Deluge system [10] was developed for reprogramming Mica2 motes. Deluge propagates software update over the ad hoc network and can switch between several images to run on the sensor nodes. An role assignment system was developed at the ETH Zurich [11] to switch between multiple tasks depending on the current requirements. Incremental network (re-)programming was studied by Jeong and Culler [12] and Panta et al. [13]. The primary focus of this work was on the delivery of software images over an ad hoc network.

We contribute to this domain by investigating techniques to upload and to replace software modules in an efficient way. In this context, efficient primarily stands for flexibility and careful resource utilization. We concentrate on lessons learned from our general software management approach for sensor networks that uses profile-matching techniques for identifying appropriate nodes for new software applications [6], [14]. This system allows, based on exchanged profiles of sensor nodes (hardware and software) and descriptions of the application's objectives, to identify modules to be installed on particular sensor nodes. The working principle is depicted in Figure 1. We implemented this system for TinyOS [15]. Unfortunately, each reprogramming step required to compile and to install a complete software image and the necessary reset of the node erased all collected status information.

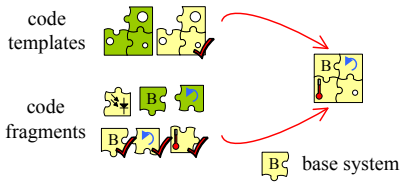

Fig. 1. Profile-matching based reprogramming architecture for sensor networks [14]

There are two popular micro controller operating systems supporting modularity: Contiki [16] and SOS [17]. Both have limitations that do not suite our needs. While SOS does not support any thread model, Contiki has a library, which maps threads to so called protothreads. Contiki makes only little use of dynamic memory management and currently supports only a single module. This limits the flexibility in adding new applications. Finally, both operating systems do not support to recover data saved in RAM after replacing the kernel.

Therefore, we started the development of an architecture that allows not only just to reprogram a sensor node but also

to do this in a per-module way. As the hardware platform, we selected the BTnode sensor node (Section II). It represents a typical sensor node as used especially in academic research projects. This selection was necessary as various micro controller require very different handling of flash memory. The used operating system is Nut/OS, a non-preemtive, cooperative multi-threaded OS for sensor nodes.

The following objectives were defined for this architecture:

- The system should be able to replace single modules including possible threads. This includes adding and removing modules to or from the system, respectively.
- The system kernel should be replaceable. This feature usually requires special handling of the flash memory.
- As a main capability, variables and dynamically allocated memory should be saved during module replacement.

These objectives already define the main contributions of this paper. We developed a system that can replace either single modules on the Nut/OS system as well as the entire kernel. To our knowledge, this is the first system that also allows to restore the local state of a module after reprogramming in such a flexible way. The developed architecture is presented in Section III and the implementation aspects are discussed in Section IV.

II. The BTnode Sensor Node

Before introducing the reprogramming architecture, we briefly outline the selected hardware platform, the BTnode, a wireless mote class sensor node developed by the ETH Zurich, and the BTnut operating system.

A. Hardware

The BTnode platform is building the basis for our approach. It is based on an Atmel ATmega128 micro controller. For wireless communication it provides a CC1000 radio chip as well as a Bluetooth radio. The Atmel ATmega128 micro controller is a RISC processor with $128\,\mathrm{KB}$ flash ROM and $4\,\mathrm{KB}$ internal SRAM that can be accessed much faster compared to external memory. The $4\,\mathrm{KB}$ internal SRAM are extended to a total of $64\,\mathrm{KB}$ SRAM. Additional $180\,\mathrm{KB}$ can be accessed by banking the memory. The Harvard architecture, i.e. program and data memory are addressed independently, uses the flash as program and the SRAM as data memory. The most important technical information is summarized in Table I.

Architecture	RISC / Harvard
Flash Memory (wite cyles)	128K (10,000)
EEPROM (wite cyles)	4K (100,000)
Internal SRAM	4K
Maximum Frequency	8MHz

TABLE I
Selected technical data for the Atmel ATmega128L

Besides programming the flash ROM using an In-System Programmer (ISP) connected to a PC, e.g. via a serial interface, the ATmega128 also supports self-programming of the flash memory. The flash is divided into two sections: a regular and a boot loader section. The flash can only be programmed from within the boot loader section, which resides in the last $8\,\mathrm{KB}$ of the flash. Furthermore, the flash is divided in pages of $256\,\mathrm{Byte}$. Before it is possible to write a page, the page must be erased. It is important to note that these two operations, write and erase, are independent from each other. Whereas a reset interrupt is delayed during a write operation, this is not the case between the erase and write operations. It is therefore possible that the node is reset after the erase leading to an erased but not written page.

During a write operation on the regular flash section, it is not possible to access or execute code from this section. This lock affects the entire flash, not only the affected flash page. This behavior must be considered for reprogramming the node using communication protocols, which require real time execution.

B. Software

The BTnode is supported by two operating systems that are well-known in the sensor networking community: TinyOS and BTnut, the latter one being the primary OS used for the BTnode. Whereas TinyOS is also heavily used in the sensor networking community, BTnut has a number of features that make it a perfect source for our reprogramming work.

BTnut is build on top of the multi-threaded Nut/OS framework. Nut/OS is a non-preemtive, cooperative multi-threaded OS, which makes extensive use of dynamic memory management. Besides supporting thread priorities, the number of threads is only limited by the available memory. Opposed to other operating systems including recent version of TinyOS, a Nut/OS thread does not need any static variables. The stack and heap as well as memory used for thread management are allocated during thread startup. Therefore, it is not necessary for the compiler to know details about the threads at compile time. Further features include POSIX-like interrupt driven streaming I/O system.

III. Reprogramming Architecture

In the following, we outline the main principles of the developed reprogramming architecture. Basically, the following objectives are addressed:

- Module replacement
- Kernel replacement
- Saving of single variables
- Saving of dynamically allocated memory

As the architecture depends to a certain extend on the particular BTnode hardware, we obviously focus on the specific capabilities. Nevertheless, the principles can be easily transferred to similar architectures that are based on the ATmega128.

A. Loading modules

This section describes the theoretical background of adding code to an already existing binary. The primary system of the node is represented by a kernel. The kernel consists the main operating system containing a scheduler, I/O interfaces, and other service functionality. Additional functionality can

be added at a later point of time in form of a module. The module must contain information for the kernel which code to execute but this will be described later in this section.

When adding code to a node several things have to be addressed. Due to the Harvard architecture, it is not possible, or at least quite expensive in terms of system performance, to modify the code once it is written to the flash. This has to be taken into account while designing module support. There are two main subjects which must be discussed when designing support for modules: How to access the kernel functions and when to do the final linking.

A function can be called in two ways. Either direct or indirect. Usually, a method is directly accessed: The code contains the call command followed by the address of the command. When calling a function via a pointer this is an indirect call. The address of the function is loaded into a special register and then the "call indirect" command is executed. This also applies to other name to address mappings like variable names. As they work equivalent, only function calls will be discussed here.

Either way, the function name must be mapped to the function address. These name to address mappings are stored in a symbol table. When linking a program, the linker first creates a symbol table and afterwards replaces all function names by their memory addresses. When linking a module, it is not possible to change the addresses of the kernel anymore as it is already programmed onto the node. Therefore, a symbol table containing the symbols of the installed kernel must be provided to the linker.

The linking itself can either be done on the host computer or the node. While linking on the node allows greater portability, e.g. because the same unlinked code can be used on nodes with different kernels and may also placed at different positions, this also has drawbacks. First, the kernel's symbol table must be provided with the kernel. Contiki states that the kernel's symbol table has a size of $4\,\mathrm{KB}$ [18]. Besides that, the unlinked file has to be provided with linking specific information. Therefore, the flash memory needed as well as the data to be transmitted increases when linking on the node.

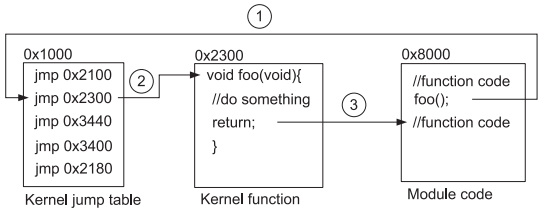

Fig. 2. Using jump tables

If it is not possible or not wanted to resole the function addresses at linking time, this can be done using a jump table. A jump table is a list of jump commands. An example can be seen in Figure 2. Instead of calling the real function, of which the address is unknown, the jump to that function in the jump table is called (1). The command in the jump table jumps to the real function (2). When a function is called, the program counter is saved on the stack. Therefore, a return jumps back to the calling function (3). The jump table and the jump to a function must always stay at the same address. Because of this, it is not possible to change the address of a kernel function, but not to add or to remove functions from the jump table. Therefore, this is only a reasonable solution if only a limited number of functions are provided by the jump table.

Using indirect calls, similar solutions can be implemented. Either by looking up the address of a function using a symbol table that is stored on the node or just by storing the address of a function in a table without the jump command. Both of these solutions are less efficient then the ones using direct calls.

Using only relative calls and jumps, it is possible to create position independent code. This code can be placed at any address. As relative jumps are limited to $\pm 4\,\mathrm{KB}$, the size of a position independent module is limited to approximately $4\,\mathrm{KB}$. If the module is bigger then $4\,\mathrm{KB}$ not only the addresses of kernel functions, but also the final position of the module must be known when linking.

Not only the address of the text section, which contains the executable code, but also the data section containing global and static variables must be set during linking. While this is not a problem when rebooting after uploading a module - the data section of the module can be put right behind the data section of the kernel - this is a problem when adding a module during runtime. There is no unused data memory available, because all memory not used by the data section of the kernel is used by the dynamic memory management.

Besides replacing code, there is the problem of saving local variables. Although it is possible to replace a thread during runtime [19], it is difficult to implement this solution efficiently on the ATmega128 micro controller. As the micro controller is not powerful enough to do the needed analysis, a huge amount of data, e.g. the stack, has to be transmitted to a host computer for analysis. It is therefore more reasonable to signal the thread to go into a safe state from which it can recover when being started again.

B. Kernel replacement

The basic approach for replacing the kernel is to temporarily store the new code in memory, flash the data appropriately, and reboot the new kernel. As there is not much difference to other reprogramming approaches that always replace the entire software image, we concentrate on the possible problems of the kernel replacement that need to be considered during the reprogramming procedure.

In our scenario, the data transmission can not be done by the bootloader and depends on a working kernel. Therefore, the new kernel must be buffered on the node as it is not possible to replace the kernel while transmitting data. On the BTnode, it is possible to either buffer the new kernel on a free space on the flash memory or in external SRAM. When the new kernel is stored in the buffer, the node reboots and the bootloader replaces the old kernel with the one saved in the buffer.

During the update procedure, two problems can bring a node in a state where it is not possible to recover, at least

not from a remote connection. Either the kernel has a bug or it gets corrupted. In case of a buggy kernel, the only way to recover the node is to have a guaranteed working copy of kernel saved on the node (Golden Image). If the node detects a buggy kernel, it can then load the Golden Image. However, for this solution to work it must be detected that the kernel is not working properly.

While it is difficult to technically avoid bugs in the kernel, it is possible to avoid the corruption (or at least to minimize the possibility of corruption). A simple way of detecting a corruption is using checksums. When a corruption is detected, the data must then be recovered, e.g. by means of retransmission. As the kernel is needed to retransmit data, it must be assured that it is possible to recover after any kind of possible failure. The most likely critical failure during flashing a new kernel is power loss. When using SRAM to buffer a new kernel, a possible power loss during the flash operation must be ruled out, e.g. by checking the current battery status.

C. Memory management

After replacing a kernel or module, it is often wanted to recover state information. The usual approach is to store data in the EEPROM. Whereas this is the appropriate place to store such as settings, the maximum number of write cycles for the EEPROM must be considered in scenarios where such operations are frequently applied. With a lifetime of about 100,000 write cycles, the lifetime is exceeded in a little more then a day if writing a value every second.

Saving data in SRAM is problematic because of possible power loss. In exchange, SRAM is faster, bigger in size, and has practically unlimited write cycles. Therefore, neither EEPROM nor flash ROM are an alternative for frequent changes of status information. It is also possible to back SRAM with a super capacitor for several days. The main challenge is to recover that status data after a reset.

IV. Implementation Details

In the following, we discuss implementation specific details of our reprogramming concept related to the BTnode architecture and the Nut/OS operating system. These aspects reveal some deeper insights into the problem of developing the dynamic reprogramming scheme for a real hardware platform.

A. Flash management

As the flash is normally not used for saving different data parts, we created a flash memory management system. We decided to use a simple linked list. Each node in the list starts with a header and is aligned to the pages. The header contains information about the contents of the node, and its length. To detect corruption, the header is equipped with a CRC32. This CRC is calculated over the whole node with the CRC field masked. This allows to detect data corruption if anything went wrong while flashing data.

There is one exception though. As the kernel must always start with the interrupt vectors, the header is placed behind the interrupt vectors. Kernels, which are saved on flash for replacement of the original kernel, get an extra header.

Besides the name of the application, it also contains the needed stack size and the entry function. This information is needed to start an application properly. The header includes the following fields:

- The length of the application
- The name of the application
- A pointer to the entry function of the application
- The starting page for which the application was compiled
- The needed stack size
- A CRC32 of the application
- Flags (deleted, kernel)

When data is overwritten, the data is first erased from back to front and then written from front to back. This allows the bootloader to easily recover data as the old header is kept as long as possible. Otherwise, the next non-empty page must contain a valid header.

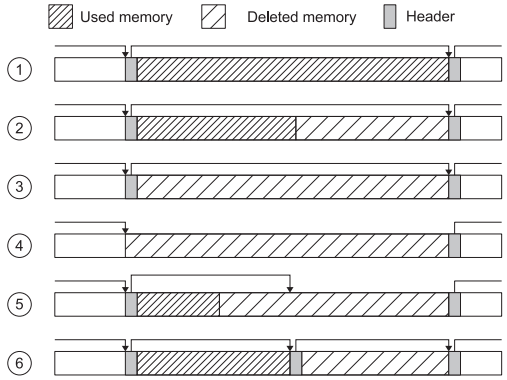

Fig. 3. Deleting and adding nodes from/to the linked list

Figure 3 outlines the usage of the linked list in detail. In step (1), there is an application saved between two other nodes. Its header points to the next node. If part of the data is deleted, the linked list is still valid as shown in step (2). The situation in step (3) can be found when a node is empty: Only the page containing the header is written. When the header is deleted in step (4), it is easy to recover the list by finding the fist page, which is not deleted. Step (5) shows the new header and part of the module written. Note that the pointer still points to a deleted page. Again this can be fixed looking for the next written page. After completely writing the application, the header marking the empty space is written as shown in step (6).

B. Creating kernels and applications

The header used for the flash management, is added at compile time using compiler attributes to put it in a special section. During the linking, it is then placed at the correct location and missing information is added using symbols. As described in Section III-A, symbols are used to replace the name of a function with an address. Similarly, instead

of passing an address this technique can be used to pass a variable.

The following expression can be used to initialize the variable `foo` with the address of the symbol `usersymbol_foo`:

```
uint16_t foo = (uint16_t)&usersymbol_foo;
```

When assigning a value to `usersymbol_foo` instead of an address and passing this to the linker, `foo` gets initialized with this value. This approach makes use of the default tool chain and avoids the manipulation of the final files.

For the CRC32 calculation all files are linked twice. For the first linking, the CRC value is set to zero. Then, the CRC32 is calculated for the binary image and the files are linked again, with the correct CRC. As pointers are only 16 Bit on the 8 Bit AVR, the CRC has to be passed to the linker using two symbols, each containing a word of the CRC. After linking the kernel, its symbol table is extracted and saved in a special directory, using the CRC as filename for later use.

In order to compile and link the application to the right kernel running on the BTnode, the kernel CRC is queried before linking the module. On the basis of the CRC, the adequate symbol table can be selected. Afterwards, the module is linked for the first time to determine it's size. Library functions that are not included in the kernel will be automatically be linked into the application as they do not show up in the symbol table. Knowing the size of the application, program memory can be allocated on the node. A second linking step is needed to place the application at the right memory address. Finally, the code has to be linked a third time to add the correct CRC. The resulting file can now be transmitted to the node, flashed, and then executed.

Currently, we are facing several limitations. So, is not possible to share functions between modules. It is therefore also not possible to upload a functions or driver library. All functions not provided by the kernel must be linked into the application. Furthermore, it is not possible to use the data section when writing applications. All data must therefore saved in the program section and be loaded at runtime.

C. Flashing

Flashing is done in two circumstances: By the kernel, when receiving an application or new kernel, and by the bootloader when replacing the kernel. To make the flash functions, which are part of the bootloader, accessible by the kernel a jump table is used. This allows the kernel to access these functions without explicit information about the bootloader.

When flashing data, all interrupts must be turned off as it's not possible to execute code outside the bootloader section while flashing. This must be considered when using teal time constraints. For example, using RS232 without flow control is problematic as it might loose data.

The ATmega128 supports moving the reset vector to the bootloader section. This will always call the boot loader after a reset. It is then the responsibility of the boot loader to start the main code. To update the kernel the node gets reset after writing the new kernel to the flash. When the bootloader starts up, it verifies the flash using the CRCs, if necessary deletes broken modules and checks whether it finds an updated kernel. To avoid having a corrupted kernel, the bootloader verifies the CRC of the new kernel before deleting the old one. It also makes sure that the new kernel does not overlap with itself when being copied. After copying the kernel, the copied kernel gets verified, before the buffered copy gets deleted to free memory.

D. Named memory

In order to recover data saved in SRAM after a reboot, we decided to use a concept similar to shared memory. Shared memory is accessed using a key. Similarly, we decided to use a string to build "named memory". It is now possible to allocate memory and assign a name to it. Later on, it is possible to get a pointer to this memory using this name - even after the node got rebooted or the kernel replaced.

Named memory is allocated like other managed memory but it is assigned an extra header. This header contains the name of the memory block, a pointer to the next named memory block, and a CRC8, which is calculated over the header. The root of this linked list is placed at the very end of the memory during the initialization of the managed memory. This way, the probability of a conflicted with a resized data section of the kernel is minimized.

During the boot process, the root element of the list is verified using the CRC. Now, it is possible to walk the linked list verifying each header. While this does not verify the content itself, it is quite unlikely that the contents are modified without touching the headers. The space in between the named memory is then freed for the normal memory management.

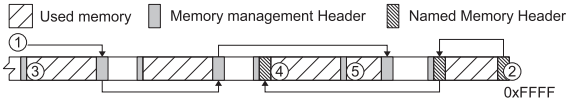

Fig. 4. Named memory

Figure 4 shows how the memory will be organized when using the concept of named memory. (1) is the linked list of free memory. (2) is the root of the linked list for the named memory. It is at the very end of the memory. Allocated memory always has a header containing the size of the node (3). (4) shows a named memory block. The memory used for this block is allocated by the normal memory management and, therefore, has its header containing the size. Behind that, the named memory header is saved. Although normally allocated memory is preferredly taken from the beginning and named memory from the end, this can still be mixed (5).

V. Proof-of-Concept Example

As a simple example, we use an applications to demonstrate the thread support of Nut/OS. It basically consists of a simple thread that repeatedly outputs a character. The following listing shows the adjusted version of this sample application. `APPLICATION` is a macro, which adds the header to the module. The parameters are the application name, the stack

size, and a pointer to pass arguments to the application. The latter is not used in the example. After being started, the program will print an "U" character to the default output, which is normally the serial port, and then sleep for 125 ms.

Listing 1. Sample application

```
APPLICATION("printU", 192, arg)
{
    NutThreadSetPriority(16);
    for (;;) {
        putchar('U');
        NutSleep(125);
    }
}
```

The size of the binary code for this application is 54 B of which 26 B are of executable code. When making adjustments to this code, 52 B have to be transmitted compared to over 20 KB when transmitting the entire kernel. While this is a small and simple application, the advantage over transmitting the entire kernel is obvious.

Once the application is stored onto the sensor node, it can be started passing "printU" to the module startup function, which will initiate a new thread and then start the code.

VI. Conclusion

In order to support more abstract higher layer reprogramming strategies in sensor networks, the efficient replacement of application modules in single sensor nodes is strongly necessary. Based on this observation, we identified the needs and objectives for lower layer code replacement. We identified three major building blocks for such reprogramming: The replacement of single modules or the entire kernel, and, most importantly, the ability to keep local state information, i.e. variables and dynamically allocated memory, during such reprogramming actions.

Based on Nut/OS we have implemented support for application modules. We demonstrated that it is possible to load and execute modules during runtime. Using a CRC32 for kernel identification, it is possible to support collaborative use of single sensor nodes without having to replace the kernel to ensure proper linking. We also developed a simple way to save and recover the module state, i.e. the content of variables, after a reset or kernel update. We named this concept "named memory", which is built on top of the normal memory management and, therefore, adjusts to needed and available resources.

Open issues primarily target the memory management. First, it is currently not possible to use a data section in the code besides using the named memory concept. An issue, which gets more severe using modules, includes memory leaks caused when terminating threads. At the moment, allocated memory is not associated to a thread and it is therefore not possible to free allocated blocks automatically. This is a flaw which has it roots in the Nut/OS memory management, but needs further investigation.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Elsevier Computer Networks*, vol. 38, pp. 393–422, 2002.

[2] D. Culler, D. Estrin, and M. B. Srivastava, "Overview of Sensor Networks," *Computer*, vol. 37, no. 8, pp. 41–49, August 2004.

[3] Y. Guo, P. Corke, G. Poulton, T. Wark, G. Bishop-Hurley, and D. Swain, "Animal Behaviour Understanding using Wireless Sensor Networks," in *31st IEEE Conference on Local Computer Networks (LCN): 1st IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, November 2006, pp. 607–614.

[4] A. Baggio, "Wireless sensor networks in precision agriculture," in *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 2005)*, Stockholm, Sweden, June 2005.

[5] I. F. Akyildiz and I. H. Kasimoglu, "Wireless Sensor and Actor Networks: Research Challenges," *Elsevier Ad Hoc Networks*, vol. 2, pp. 351–367, October 2004.

[6] 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 Conference on Communication System Software and Middleware (IEEE/ACM COMSWARE 2007): 2nd IEEE/ACM International Workshop on Software for Sensor Networks (IEEE/ACM SensorWare 2007)*. Bangalore, India: IEEE, January 2007.

[7] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava, "Sensor Network Software Update Management: A Survey," *ACM International Journal on Network Management*, vol. 15, no. 4, pp. 283–294, July 2005.

[8] C. Margi, "A Survey on Networking, Sensor Processing and System Aspects of Sensor Networks," University of California, Santa Cruz," Report, February 2003.

[9] D. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo, "A Network-Centric Approach to Embedded Software for Tiny Devices," in *First International Workshop on Embedded Software (EMSOFT 2001)*, Tahoe City, CA, USA, October 2001.

[10] A. Chlipala, J. Hui, and G. Tolle, "Deluge: Data Dissemination for Network Reprogramming at Scale," University of California, Berkeley, Tech. Rep., 2004.

[11] C. Frank and K. Römer, "Algorithms for Generic Role Assignment in Wireless Sensor Networks," in *3rd ACM Conference on Embedded Networked Sensor Systems (ACM SenSys 2005)*, San Diego, CA, USA, November 2005.

[12] J. Jeong and D. Culler, "Incremental Network Programming for Wireless Sensors," in *First IEEE International Conference on Sensor and Ad hoc Communications and Networks (IEEE SECON)*, June 2004.

[13] R. K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," in *26th IEEE Conference on Computer Communications (IEEE INFOCOM 2007)*, Anchorage, AK, May 2007.

[14] F. Dressler, G. Fuchs, S. Truchat, Z. Yao, Z. Lu, and H. Marquardt, "Profile-Matching Techniques for On-demand Software Management in Sensor Networks," *EURASIP Journal on Wireless Communications and Networking (JWCN), Special Issue on Mobile Multi-Hop Ad Hoc Networks: from theory to reality*, vol. 2007, no. Article ID 80619, p. 10, 2007.

[15] Z. Yao, Z. Lu, H. Marquardt, G. Fuchs, S. Truchat, and F. Dressler, "On-demand Software Management in Sensor Networks using Profiling Techniques," in *7th ACM International Symposium on Mobile Ad Hoc Networking and Computing (ACM Mobihoc 2006): 2nd ACM International Workshop on Multi-hop Ad Hoc Networks: from theory to reality 2006 (ACM REALMAN 2006), Demo Session*. Florence, Italy: ACM, May 2006, pp. 113–115.

[16] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *29th IEEE International Conference on Local Computer Networks (LCN)*, Tampa, FL, USA, November 2004, pp. 455–462.

[17] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *3rd ACM International Conference on Mobile Systems, Applications, and Services (ACM MobiSys 2005)*. Seattle, WA, USA: ACM Press, June 2005, pp. 163–176.

[18] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *4th ACM Conference on Embedded Networked Sensor Systems (ACM SenSys 2006)*. Boulder, Colorado, USA: ACM, November 2006, pp. 15–28.

[19] M. Felser, R. Kapitza, J. Kleinöder, and W. Schröder-Preikschat, "Dynamic Software Update of Resource-Constrained Distributed EmbeddedSystems," in *IFIP International Embedded Systems Symposium (IESS'07)*, vol. 231/2007, Irvine, CA, USA, May 2007, pp. 387–400.