Supporting Application-Oriented Kernel Functionality for Resource Constrained Wireless Sensor Nodes

Hyojeong Shin and Hojung Cha

Department of Computer Science
Yonsei University, Seoul, Korea
{hjshin, hjcha}@cs.yonsei.ac.kr

Abstract. A sensor network application requires diverse kernel supports to function properly. With its resource limits the sensor node cannot provide all the functionalities needed by many kinds of applications at the same time. The kernel's functionality therefore requires runtime reconfigurability, which can be achieved via modularizing the kernel. This paper presents a framework that dynamically reconfigures the kernel's functionality according to the needs of the application. In particular, the proposed mechanism handles the address resolution problem of a MMU-less processor. This framework has been implemented on a sensor network operating system, RETOS, which supports multi-threaded programming environments. It efficiently manages the modularized kernel's resources and works in an optimized condition. By providing modularized kernel programming, RETOS optimizes itself with functionalities that various kinds of sensor network applications require.

1 Introduction

Diverse applications exist in wireless sensor networks; for example, acoustic source localization, environmental monitoring, mobile object tracking, etc. System functions such as localization, time synchronization and data aggregation are required to make such applications work, so the sensor operating system should support them. A sensor OS should have general-purpose operating system features to provide these diverse services efficiently to applications. It is hard to provide all the functionalities necessary for a sensor network operating system at one time, because sensor nodes have limited energy, small memory and low computational power. Providing diverse functionalities to the application layer is therefore one of the important issues of a sensor network operating system. This issue is redefined as the problem of adequately selecting an operating system's functionalities, given the particular application. To implement such features efficiently, current approaches modify and re-distribute the operating system [1], or separate an application from the system to reduce the cost of reprogramming [2] [3]. Modifying and redistributing a system incurs the overhead cost of updating the whole kernel code. The modular approach, which separates the application from the system, can easily modify the functions for the applications' needs. However, this modification is done in the framework that the operating system supports. It implies that the modification is done only at the application level.

Considering the resource constraints of sensor network hardware, this paper proposes a framework that provides a mechanism to support diverse kernel functionalities that are needed only by the application at hand. The self-reconfiguration is achieved by selecting the appropriate kernel components in the operating system without changing the application. The proposed framework provides such a dynamic code update that means kernel level adaptability. This feature supports the multi-level network stack which provides application-aware routing protocols [4]. The network stack transmits a packet in various ways according to the application's requirement. The system has been implemented on RETOS [4] [5], which is a sensor network operating system that provides module-based multi-threaded programming. A loadable kernel module system is implemented as a part of the RETOS kernel to support modifying the kernel functionality.

The rest of this paper is structured as follows. Section 2 reviews the background related to our work. Section 3 presents the proposed system's key mechanisms. The system is evaluated in Section 4. Finally, Section 5 concludes the paper.

2 Background

2.1 Function Reconfigurability

Ideally a sensor network operating system should provide reconfiguration of the system functionalities as required by applications. TinyOS [1], the de facto operating system for sensor nodes, supports modularized programs, which are compiled into a monolithic image and distributed as a whole. Maté [3] implements a simple virtual machine (VM) on the system that allows the building of an executable and updatable code to enable modularized updating. Similarly, Magnet OS [6] builds a Java VM on the system. A good example of supporting modularized programming in sensor networks is SOS [2], which uses a text address and a stack base to indicate the addresses of variables and functions used in a binary code to handle the problem of MMU-less microprocessors. Contiki [7], a thread-based operating system for sensor networks, provides code update in terms of independent applications. The VM*[8] serves as middleware programming, which allows the building of a program for VM and a customized VM for a corresponding binary.

Reconfiguring the kernel functionality originates from traditional embedded systems. uClinux [9], an embedded version of Linux [10] ported on an MMU-less microcontroller, supports a variety of Linux functionalities and a loadable kernel module. With a relocation mechanism and Position Independent Code (PIC), uClinux builds a code that runs independently from a position in the allocated memory. uClinux successfully extends the system functionalities in general-purpose embedded systems. However, it cannot be used directly in sensor networks due to the sensor nodes' resource constraints.

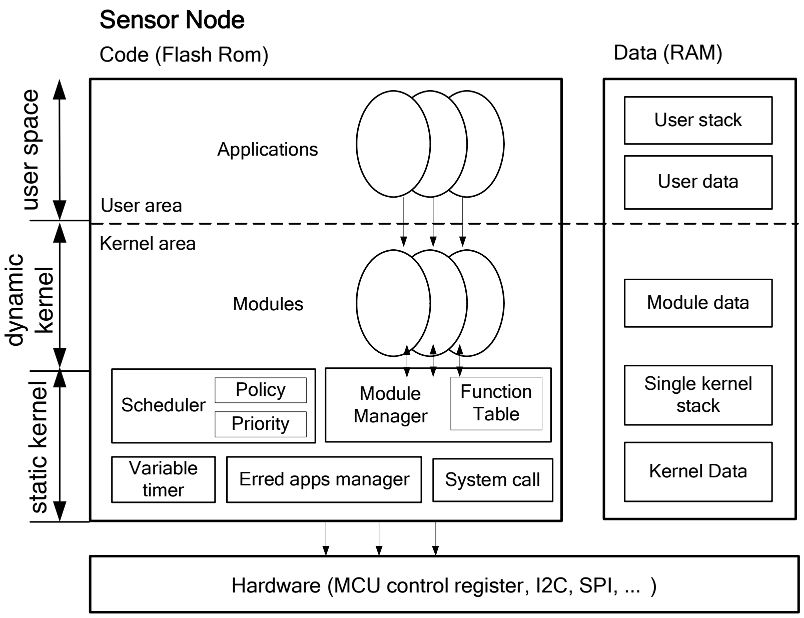


Fig. 1. RETOS overview

2.2 RETOS Overview

RETOS [5] is currently being developed to support a reliable and multi-threaded programming environment for sensor networks. It has three objectives: (i) to provide a multi-threaded programming environment, (ii) to serve as a reliable system which protects a kernel from errant applications, and (iii) to flexibly extend the sensor operating system's functionalities. For the multi-threaded programming environment, RETOS provides the POSIX 1003.1b real-time scheduler interface. With a dual mode operation and a protection mechanism, RETOS can protect the kernel from errant applications. The RETOS kernel shares the kernel stack among its threads to reduce memory usage incurred by dual mode operation. The application, kernel and kernel modules are developed in standard C language. RETOS is supported by several types of hardware including Telos Skymote [11] which is used in the evaluation part.

To build a flexible system, RETOS provides dynamic application installation. Moreover, RETOS supports a Loadable Kernel Module (LKM) that modularizes system functionalities as kernel modules and dynamically replaces them. This feature provides system level reconfiguration to support flexibility to diverse applications' operations. Figure 1 shows an overview of the RETOS system that features the Loadable Kernel Module. An application is loaded onto the system after the safety check, and system resources are then allocated. The operating system consists of two parts: a static kernel which includes a core part of the system and hardware dependent codes, and a dynamic kernel which includes common libraries used by applications and various kernel services. Section 3 presents the proposed RETOS LKM system and corresponding mechanisms.

3 Loadable Module Support

The LKM framework consists of the module manager that registers a modularized kernel on the system, and the function table that links functions of modules to applications. The following section describes the technical issues that arise in modularizing the kernel.

3.1 Kernel Memory Management

A binary code in general consists of text, heap, stack and data area. Some of the recent embedded systems support the eXecute In Place (XIP) technique to execute code on flash memory without loading it to the memory space, hence allocating memory for the text is not required. The amount of dynamic allocation is unpredictable and depends on the types of application, thus the heap size is hard to determine in the compiling time. This section presents a method for handling the stack and the global variable data.

The stack size is difficult to estimate and differs for each application; therefore, the stack space should be sufficiently reserved. Because much redundancy occurs in the stack space when using a global fixed stack for all applications, the stack size is specifically set depending on the application. The stack size is, however, eventually fixed, and results in some redundancy in the stack. To reduce this redundancy the RETOS LKMs share a common stack area with each other. The LKM uses the kernel stack as its common stack area. Every kernel module uses the single kernel stack to save intermediate results, thus the system benefits from saving redundant memory space. Figure 2 illustrates a snapshot of the module's memory usage in runtime. Each module has its own data area, but they all share the stack. During a module's execution the system prohibits another module from entering execution mode to prevent damage to previous contexts in the stack or any changes in the stack size.

To maintain the modules' information and status, memory resources are required to save them. Symbol table and memory allocation are usually used for storing the modules' information and status. A symbol table, managed by the kernel, is a manager that handles variables used in the modules. The symbol table's size depends on the number of variables used in the kernel module, which is unpredictable and should be controlled by the kernel. Also, overhead is incurred when accessing variables from the symbol table upon a system call. On the other hand, using memory space to allocate data used in the kernel module is a simple and general approach. This method is used in our work. The module's information and status are declared as global variables of the module and can be compiled with the module source. The global variables are allocated in the bss and data area of the memory space allocated to the kernel module.

The framework proposed in this paper allocates a memory space to maintain the module's information. The kernel module is compiled with a given stack base and data space address. The addresses of the functions and variables are decided a priori based on the address given. These addresses change when the kernel module is saved in flash memory and loaded into the RAM. Therefore, a mechanism is required to handle such indeterminate address changes. Figure 2 shows that each module has its own data space. Here, each kernel module is compiled separately with its own data space allowing direct access.

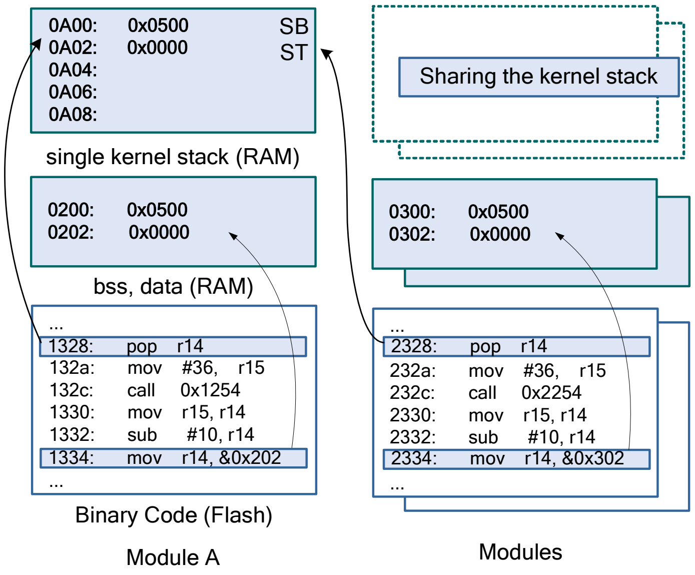
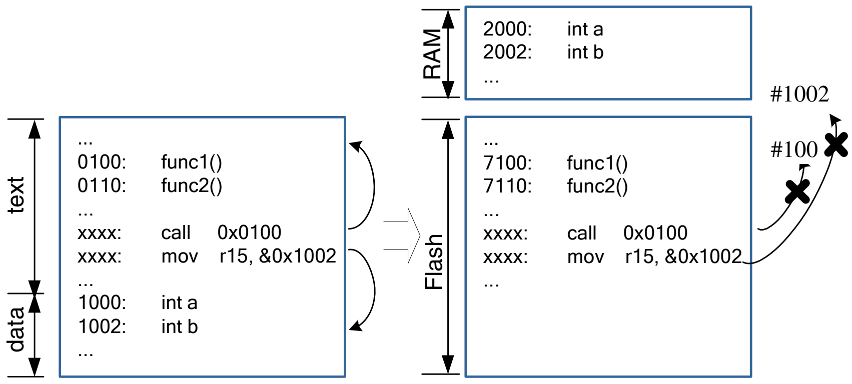


Fig. 2. Memory addressing

3.2 Dynamic Module Linking

Sensor node hardware typically uses a MMU-less microprocessor. The MMU translates a virtual address into physical memory and controls the memory's access privileges. A TinyOS-based application's binary image is always allocated in the same position in the memory space, thus no explicit memory management is needed. However, the location of the modularized code, which is used by module supporting operating systems such as SOS and RETOS, loaded in the memory cannot be expected. This means that the variables' and functions' addresses can be changed at load time, which causes problems when calling functions and accessing variables with an incorrect address. Figure 3 illustrates this problem. Figure 3 (a) shows a binary code after the compilation of the source code using the given stack base and text area address. The code accesses a function located in address number 0x0100 and a variable located in 0x1002. Figure 3 (b) shows the code whose text data are located in address number 0x7000 in the flash memory and whose data are located in address number 0x2000 in the RAM. Without any appropriate modification, this binary code may access the wrong address. This paper considers two approaches to preventing such a problem: PIC and address relocation.

Position Independent Code is a compiled code that does not have an absolute address and therefore can be run without knowing where the module is located. Function calls and jumps within the single binary code are compiled using PC-relative instructions, and variable access is compiled through indirect accessing. The compiled code runs without any code modification in run-time. PIC causes run-time overhead when modifying the Data Base Section Register by indirectly accessing a variable's specific address. SOS compiles the module using PIC. Meanwhile, the relocation method compiles the source code, assuming the location of the binary code is zero, and modifies every address accessed in the binary code when the assumed address changes.



(a) after compilation (b) after installation

Fig. 3. Linking problem

Both PIC and the address relocation would cause system overhead in different time domains. PIC generates a relative-address code that does not need to be changed, although it has run-time overhead. Using relocation, however, an absolute-address code is generated, but overhead exists to maintain a relocation table and to modify the module. Our experiment shows that the PIC-based approach requires more overhead to access the functions of the module in a sensor node than the address-relocated approach. Under the assumption that run-time overhead is more important than loading overhead in wireless sensor network applications, RETOS uses relocation to handle the indeterminate memory address problem. RETOS acquires the kernel module via a relocation table from a host, and performs the relocation in run-time.

The uClinux system executes the relocation process when the binary stored in flash memory is loaded into the RAM. The code image on the flash memory contains a memory access that references an illegal position, so the code cannot be executed in flash memory. XIP technique is very important in terms of memory efficiency in memory limited sensor nodes, since it does not require extra memory for *text* area. RETOS stores kernel code in flash memory after the relocation process. Thus,

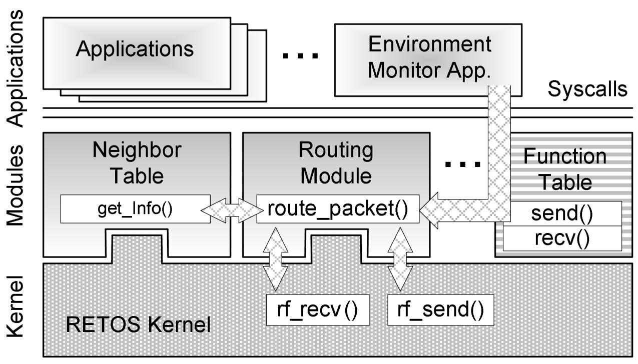


Fig. 4. Module accessing

RETOS can execute a code in flash memory while supporting relocation, whereas uClinux cannot provide such a service.

Relocation is also supported in uClinux. uClinux relocates the module when it is loaded into the RAM. Because the binary image on the flash memory contains illegal address accesses, uClinux does not support XIP using relocation. RETOS relocates the module both on the RAM and on the flash memory and the module runs on the flash memory. This reduces memory usage and the loading cost of an application. It is suitable for resource constraint systems such as wireless sensor networks.

3.3 Module Communication

RETOS has a function table that allows modules and applications to access other module's functions. The function table manages the function information of modules, such as function entry points, ownership, parameters and return types, which are accessible by other modules or applications. The module registers, un-registers and accesses the functions through the function table. The kernel and kernel modules are dynamically linked at run time even though they are directly linked. The dynamically loaded module is allocated with a kernel code in flash memory as a single executable code image, which is also dynamically removable. An application can also access a module's function, but it is done by calling a specific system call. The cost for invoking such a module's function is the same as other conventional system calls.

Figure 4 shows how a module and an application access a module's function through the function table. The application accesses the module to send a packet through radio. To perform the operation, the routing module gets the neighbor's information and accesses the kernel's hardware driver. These modules are already installed in a system and its functions are registered in the function table. In running the system, the kernel and modules work as a single image, so they access each other's functions directly. And the application accesses the module's function through a system call which references the function table and invokes the required functions. Here, the invoked system call performs a mode switch and verifies the validity of the corresponding function. This provides safety for the kernel and kernel modules from an application's illegal memory access. For kernel safety, RETOS supports dual mode operation. An application operates in user mode and a kernel module in kernel mode. While an application executes a function of the provided kernel module through a system call, the system switches the application stack to the kernel stack. These features protect the kernel from errant code [5].

3.4 Kernel Reconfiguration

A reconfigurable system reconfigures the system according to the application which runs on the system. For example, RETOS supports various routing mechanisms and selects a mechanism depending on the node's status and the current network's condition. When a node experiences low battery power, RETOS can switch the current protocol with a low-power routing algorithm. Also, when a node receives important and time-critical data, it adopts a time-sensitive routing algorithm to efficiently route the data to a sink node within the time constraint. Such a reconfiguration of the kernel functionality is supported by the modular kernel system of RETOS, LKM.

The RETOS LKM system supports kernel-level optimizing techniques. First, the system maintains a light-weight system to reduce overhead and resource usage. LKM unloads unnecessary kernel modules and only maintains modules that are required by the current working applications. The system can stop an unused system thread and reuse the resource for other applications later. Second, LKM reduces the cost for acquiring the required module over the network. RETOS supports dynamic application installment and each application requires different modules. Thus, the LKM system needs to acquire modules that are required by the installed application. Furthermore, to maintain an optimized system, the system must always keep a minimal requirement scheme by removing unused modules and acquiring newly required modules over the network with the change of application reconfiguration, which changes the requirements of the kernel functionality. This procedure needs extra network transmission and overhead.

When developing kernel modules for RETOS, the system's stability should be considered. First, the kernel module operates in kernel mode and therefore it should conduct garbage collection as resource leak in the module is critical to the whole system. Second, the module's function should return as quickly as possible. During the operating of the module function, no function in the module or application can be invoked. The module should avoid the use of loops or time-consuming logic. Last, the module should not use blocking functions because they lead to invoking context switches. By using a common shared kernel stack, context switching within the kernel module is apt to elicit unpredictable defects.

4 Evaluation

The RETOS version 0.86 is used to implement a loadable kernel module. RETOS provides multi-threaded programming, a dual mode operation and a code safety check. Each kernel, kernel's module and application is code-separated and resides in separate memory space. The experiment is conducted on the MSP430-based [11] Telos Skymote [12] hardware platform. We evaluate the performance of the RETOS LKM system in terms of system throughput and module updating cost.

For the system throughput measurement, we compared RETOS LKM against SOS, which is developed with a similar concept. For a fair comparison, two systems should be evaluated using the same hardware. RETOS is developed on msp430-based hardware. However, there is no SOS version available for msp430-based hardware. For a solution, we compared each linking method used by RETOS and SOS on Telos. Two sample applications were prepared. One was the original surge application of RETOS, whereas the other was a surge application built with position independent code, which is used by SOS. Both were executed using RETOS on msp430-based hardware. The comparison only includes the difference of overhead between the two types of linking methods, and not the operating system's overhead. With this comparison, the relocation-based system has better throughput against the PIC-based application. This comes from memory-accessing cost. Therefore, we also measured memory-accessing cost on two different types of hardware: msp-430 and AVR[13]. The module updating cost measurement was also evaluated using the surge application on three operating systems: TinyOS, SOS and RETOS. The cost for module transmission time and the installation time was measured. TinyOS uses Deluge [14] for code update, which is a well-known code updating protocol for wireless sensor operating systems. SOS and RETOS have their own module dissemination mechanisms. We installed each operating system with the surge application. To evaluate the cost for dynamic code updates, we modified a part of the application's network and measured the expenses for updating the modified parts over the network.

Table 1 shows the surge application's execution time for each operation, which is generated by a PIC compiler and a relocation linker. A relocation-produced application performs slightly faster than a position independent code based application. It only reflects binary execution overheads. The difference between the two systems is based on the difference of memory accessing and function invocation cost. A binary produced using the relocation method consists of direct addressing operations, whereas a binary using PIC consists of indirect addressing operations. As shown in Tables 2 and 3, PIC's function call and memory accessing require up to six more cycles than the relocated binary on msp430 and AVR. These overheads are generated due to modifying the base address of the position independent code and the execution of indirect addressing operations. RETOS implements direct addressing operations, hence it performs better.

RETOS supports a kernel optimization mechanism. The optimization is obtained by reconfiguring kernel compositions according to the new requirements of applications that are dynamically loaded to the deployed sensor motes. As a specific optimization operation, such a code updating mechanism is accompanied with some overheads. To measure the overhead, we modified a part of the network routing module in the surge application for each of the three operating systems. As shown

Table 1. Cost of the surge application

Cost	Relocation	PIC
Send the beacon message	0.3 msec	0.35 msec
Handle the received message	0.03 msec	0.05 msec
Updating the neighbor table	0.01 msec	0.015 msec
Active time (in a minute)	0.44%	0.58%

Table 2. Cost of accessing memory (msp430)

Operation	Relocation	PIC
Function call	5 cycles	9 cycles
Data Accessing	3 cycles	5 cycles
Global function accessing	9 cycles	9 cycles
Global data accessing	5 cycles	5 cycles

Table 3. Cost of accessing memory (AVR)

Operation	Relocation	PIC
Function call	4 cycles	10 cycles
Data accessing	2 cycles	3 cycles
Global function accessing	6 cycles	6 cycles
Global data accessing	3 cycles	3 cycles

Table 4. Data size (bytes)

Data size	Deluge	RETOS	SOS
Transmitted data size	31000	2614	2496
Flash Memory data size	31000	26542	33944

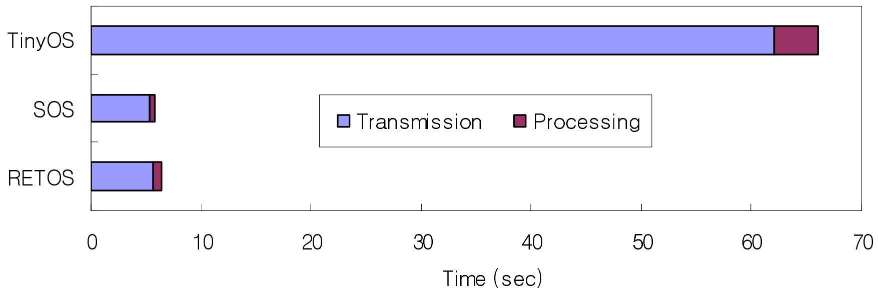


Fig. 5. Execution time of the code update

in Table 4, RETOS's updating size is smaller than that of Deluge and is similar to that of SOS, since RETOS also updates only the modified modules like SOS. Figure 5 shows the amount of time consumed for updating a module code. It consists of the transmitting time of the kernel code and the installation time for the linking process. As a small code transmission, RETOS has a smaller transmission time and installation time compared to Deluge. RETOS needs a relocation table for loading modules. The transmitted packet size is larger than the packets transmitted using SOS and also the relocation process needs extra linking process on a mote. Thus, RETOS takes slightly more time for updating modules compared to SOS. This overhead occurs only during loading a module and is negligible during run time.

5 Conclusion

The eventual goal of RETOS is to provide a general-purpose operating system for wireless sensor networks. Supporting modularized application programming suits such a goal. RETOS provides diverse kernel functionalities by supporting loadable kernel modules. The LKM system reconfigures the kernels' services according to the applications' requirements. Traditional reconfiguration approaches provide functional change by reprogramming the application; however, the RETOS LKM provides an OS level optimization for each node's applications, so it is a more general and abstract approach. The experiment showed that the LKM has a small overhead for operating the module manager. This overhead is due mainly to managing the loadable kernel system and to applying relocation to the module. In an environment where many applications work simultaneously, RETOS has the benefits of saving storage by sharing the kernel module, while reducing the LKM's operating cost.

The RETOS LKM is a first step toward an adaptable operating system. Its automatic reconfiguration mechanism includes protocols for code dissemination, and the handshaking of RETOS remains as a future work. Deluge [14], Trickle [15] and

MNP [16], which are well designed code dissemination protocols, are adaptable for RETOS's kernel module. The RETOS's module dissemination protocol must be suitable for small module kernels and customized to dynamic changes. A RETOS application may run without any consideration of the kernel's composition using the RETOS LKM system. The module resides in the kernel and can access and modify the kernel's information. The LKM system will have a mechanism to protect the system from unexpected kernel action.

Acknowledgements

This work was supported in part by the National Research Laboratory (NRL) program of the Korea Science and Engineering Foundation (2005-01352), and the ITRC programs (MMRC) of IITA, Korea.

References

[1] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister, "System architecture directions for network sensors," *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems,* Cambridge, Massachusetts, USA, November 2000.
[2] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler and Mani Srivastava, "A Dynamic Operating System for Sensor Nodes," *Proceedings of the 3rd international conference on Mobile systems, applications, and services,* Seattle, Washington, USA, 2005.
[3] Philip Levis and David Culler, "Maté: A Tiny Virtual Machine for Sensor Networks," *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems,* San Jose, California, USA, 2002.
[4] Suckwon Choi and Hojung Cha, "Application-Centric Networking Framework for Wireless Sensor Nodes," *Proceedings of the 3rd Annual International Conference on Mobile and Ubiquitous Systems: Networks and Services,* San Jose, USA, July 2006.
[5] Hyoseung Kim and Hojung Cha, "Towards a Reliable Operating System for Wireless Sensor Networks," *Proceedings of the USENIX Annual Technical Conference: Systems Practice & Experience Track,* Boston, Massachusetts, USA, May 2006.
[6] Rimon Barr, John C. Bicket, Daniel S. Dantas, Bowei Du, T. W. Danny Kim, Bing Zhou, Emin Gün Sirer, "On the need for system-level support for ad hoc and sensor networks," *ACM SIGOPS Operating Systems Review,* vol.36 no.2, pp.1-5, April 2002.
[7] A. Dunkels, B. Gronvall and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks,* Tampa, Florida, USA, 2004.
[8] Joel Koshy and Raju Pandey, "VM*: Synthesizing Scalable Runtime Environments for Sensor Networks," *Proceedings of the 3rd international conference on Embedded networked sensor systems,* San Diego, California, USA, 2005.
[9] Linux, http://www.linux.org.
[10] uClinux, http://www.uclinux.org.
[11] msp430, http://www.ti.com/msp430
[12] Tmote Sky, http://www.moteiv.com.
[13] AVR, http://www.atmel.com/products/AVR

[14] Jonathan W. Hui and David Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," *Proceedings of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, Baltimore, Maryland, USA, 2004.

[15] Philip Levis, Neil Patel, Scott Shenker and David Culler, "Trickle: A Self-Regulating Algorithm for Code Propagation and maintenance in Wireless Sensor Networks," *Proceedings of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, San Francisco, California, USA, 2004.

[16] S. S. Kulkarni and Limin Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS)*. Columbus, OH, USA, June 2005.