RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks

Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin,
Jaehyun Yoo, Chanmin Yoon
Department of Computer Science
Yonsei University
Seoul 120-749, Korea
{hjcha,sukwon,inukj,hskim,hjshin,jhyoo,cmyoon}@cs.yonsei.ac.kr

ABSTRACT
This paper presents the design principles, implementation, and evaluation of the RETOS operating system which is specifically developed for micro sensor nodes. RETOS has four distinct objectives, which are to provide (1) a multithreaded programming interface, (2) system resiliency, (3) kernel expandability with dynamic reconfiguration, and (4) WSN-oriented network abstraction. RETOS is a multithreaded operating system, hence it provides the commonly used thread model of programming interface to developers. We have used various implementation techniques to optimize the performance and resource usage of multithreading. RETOS also provides software solutions to separate kernel from user applications, and supports their robust execution on MMU-less hardware. The RETOS kernel can be dynamically reconfigured, via loadable kernel framework, so a application-optimized and resource-efficient kernel is constructed. Finally, the networking architecture in RETOS is designed with a layering concept to provide WSN-specific network abstraction. RETOS currently supports Atmel ATmega128, TI MSP430, and Chipcon CC2430 family of microcontrollers. Several real-world WSN applications are developed for RETOS and the overall evaluation of the systems is described in the paper.

Categories and Subject Descriptors
D.4.7 [**Operating systems**]: Organization and Design

General Terms
Design, Experimentation, Measurement, Performance

Keywords
Wireless Sensor Network, Operating Systems, Multithreading

1. INTRODUCTION
With the promise of wireless sensor network (WSN) applications in the ubiquitous computing era, active research has recently been conducted in a wide spectrum of devices, system software, and applications. The technology advance in WSN is primarily accelerated by the readily-available hardware platforms, as well as core system software such as sensor node operating systems in particular. From the early days, much effort has been given to develop efficient and yet complete operating systems for micro sensor nodes. For instance, TinyOS [1] has historically been used by many practitioners in the field and even advocated as "the" operating system for WSN. Other operating systems such as SOS[2], Contiki[3], MANTIS[4], and t-kernel[5] have challenged the success of TinyOS and provide incremental, or even alternative solutions for many practical issues still being debated in the related communities.

Writing an operating system for micro sensor platforms poses several unprecedented problems. First, the OS implementation should consider microcontrollers which typically provide very limited processing power, memory and battery life-time. The event-driven paradigm for sensor OS is especially favored for the resource-constrained environment, and we have seen the proliferation of TinyOS or SOS in this context. Second, the application programming model should be seriously considered to provide an easy and convenient programming interface to application developers, without needing to be aware of underlying operating system principles. The event-driven operating systems, for example, enforce programmers to structure and program an application as a state machine in terms of tasks and event handlers. Understanding this concept is an easy task for experts, but conventional programmers who are accustomed to a process model of programming may find the concept hard to grasp. Third, a micro sensor node usually does not have memory management unit (MMU) hardware. The MMU-less hardware imposes severe restrictions in implementing protected mode of OS operations. Any malicious or erroneous application program can easily disrupt other applications or even crash the kernel, because of the lack of memory protection by hardware. To provide robustness at the OS level, an effective mechanism should be devised, preferably by software. Fourth, the limited memory of only a few kilobytes in a sensor node necessitates the minimal implementation of sensor OS by reducing kernel functionality However, a wide variety of sensor applications are found in practice, and they may require different OS support, depending on the nature of the application, which is sometimes exclusive to the application. The sensor OS surely cannot provide all the functionality required by many applications at one time. The OS should therefore have a mechanism to reconfigure the kernel appropriately upon the application's request. Fifth, the networking architecture in WSN is differentiated from traditional IP-based networks in many ways. A typical sensor network application

IPSN'07, April 25-27, 2007, Cambridge, Massachusetts, U.S.A.

requires data-centric [6], rather than address-based, networking support, based on energy-efficient implementation of network and link layer protocols, over performance-constrained RF hardware. The conventional layering architecture for an IP network, for instance, is considered heavy-weight and unsuitable for WSN applications. A sensor OS should therefore provide a specifically-designed network abstraction to application programmer, and the kernel should provide an optimal implementation.

In this paper, we describe a sensor network operating system, RETOS, which has been developed to cover the various issues discussed above. Knowing that there are already a few sensor OS, some of which are mature and some which are still in the development stage, we have had specific goals to develop yet another OS for WSN. Our primary objective is to develop *a robust, reconfigurable, and resource-efficient multithreaded operating system* for off-the-shelf micro sensor nodes. The overall concept is illustrated in Figure 1. Although the event-driven approach is commonly adopted for sensor OS mainly due to its efficient implementation in a resource-constrained hardware environment, application developers manage the states of tasks and events explicitly, via program split process. We believe that, in order for a sensor network to become more popular in real world applications, the programming model should be more user-friendly and perhaps the well-understood multithreading approach could be an alternative solution, as far as its efficient implementation is guaranteed. We describe our solution for this issue. Another important fact regarding most of the currently available sensor OS is that the robustness issue on kernel and application use is barely considered in the implementation, because of MMU-less hardware. We believe that a sensor OS should guarantee a robust execution of both kernel and application. RETOS provides a software solution for this issue. Kernel optimization with adequate functionality is also an important requirement of sensor OS. RETOS explicitly separates applications from the kernel, hence an application is dynamically loaded into the system, so does the kernel module. RETOS achieves the kernel reconfigurability via a loadable module framework.

RETOS is fully functional as a kernel, and a set of real applications has been developed to evaluate the system. RETOS is initially implemented on the TI's MSP430-based motes such as Tmote Sky [7]. Presently the operating system supports ATmega128-based MicaZ [8], and even the latest CC2430 [9] SoC processor from Chipcon. As far as we are aware, RETOS is the first sensor OS which runs on this variety of microcontrollers.

The next three sections discuss the key principles of RETOS and their implementation techniques, from the viewpoint of system resiliency, multithreading support, and kernel reconfigurability. Section 5 and 6, then, describe the network abstraction provided by RETOS, and the network monitoring effort accompanied by the operating system, respectively. Some of the implementation issues on various hardware are discussed in Section 7. Overall evaluation of RETOS, rather than detailed analysis of the individual OS component, is given in Section 8 with three real-world WSN applications. We conclude the paper with related work and closing comments in Section 9 and Section 10. Note that details on from Section 2 to Section 6 are covered in our previous work. This paper reorganizes and highlights the key concept of RETOS from the overall system point of view.

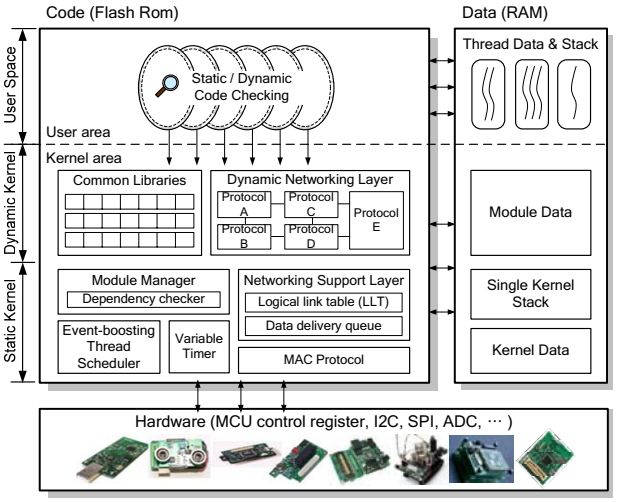


Figure1. RETOS architecture

2. SYSTEM RESILIENCY

The primary objective of a robust and resilient operating system is to enable a sensor node to run efficiently, safe from errant applications without extra hardware support. The kernel should be able to detect harmful attempts by applications on system safety, and terminate them appropriately. The microcontroller used for a sensor node typically has a single address space due to the MMU-less hardware, hence the kernel and applications exist in the same address space. The operating system mechanism for error-free wireless sensor networks should therefore operate with software assistance.

RETOS ensures system resilience with two techniques: dual mode operation and application code checking [10]. Dual mode operation logically separates the kernel and the user execution area. Application code checking evaluates the validity of compiled code via static analysis, and run-time behavior of application code via dynamic check.

2.1 Dual mode operation

In RETOS, dual mode operation is implemented by stack switching. Applications in the user mode use the user stack, and the stack is changed to the kernel stack upon system calls and interrupts handling. Dual mode operation may incur memory overhead on resource-constraint sensor nodes due to the per-thread kernel stack. To save memory usage, RETOS maintains a single kernel stack in the system. Thread switching is performed right before returning to user mode, that is, the time when all work pushed on the kernel stack is finished. Although the single kernel stack is unable to preempt threads in the kernel mode, it enables memory efficient implementation of dual mode operation.

Unlike dual mode operation on general-purpose operating systems, RETOS should implement it without hardware supports, because microcontrollers such as MSP430, ATmega128, and CC2430 do not have a privileged mode of operation. Upon an invoked interrupt, the microcontroller saves the status registers in the current stack and calls the corresponding interrupt handler. The handler function saves the current stack pointer in the TCB (Thread Control Block) and switches to the kernel stack, if the system was in the user mode. In the real implementation, some

registers might be saved in the user stack, due to stack switching, thereby leaving them modified by other threads. RETOS stores the registers within the TCB and restores them before returning to the user mode. The system call is implemented by a function call, so the return address remains in the user stack. RETOS also stores the return address in TCB and validates it.

2.2 Application code checking

In order to compromise the lack of virtual memory functionality due to the MMU-less microcontroller, RETOS provides a software technique called application code checking which consists of static and dynamic checks. Application code checking prevents user applications from accessing memory outside of its legal boundary and direct hardware manipulation. To achieve this goal, the technique inspects the destination field of machine instructions. The source field of instructions can also be examined to prevent the application from reading kernel or other applications data. Static code checking verifies direct or immediate addressing instructions, pc-relative jumps and *eint*/*dint* during the compile time. Dynamic code checking verifies the correct usage of indirect addressing instructions in runtime. Dynamic checking is also required for the *ret* instruction as the return address can be affected by buffer overrun.

Figure 2 shows the sequence of constructing trusted application code. Applications loaded on the system are allowed to store data and to execute codes in their own resources. Every source code of the application is compiled to assembly code; then checking code is inserted to the place where dynamic code checking is required. After dynamic code insertion, a binary image is created via compiling and linking, and the static code checking is then conducted on the binary. Application errors that are not detected at the compile time are reported to the kernel. When the errors are reported, the kernel informs users of the illegal instruction address and safely terminates the program.

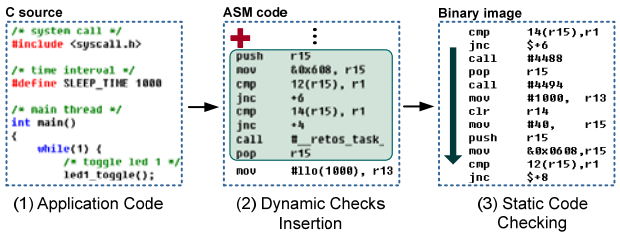


Figure 2. Generating trusted code

3. MULTITHREADING SYSTEM

In contrast to an event-driven operating system, multithreading inherently provides high concurrency with preemption and blocking I/Os characteristics of the underlying system. The application programmers may concentrate on the semantics of the problem at hand; not worrying about the optimal execution of their programs, via explicit concurrency control, as in the event-driven programming environment. Although the multithreading approach is attractive in sensor application developments, its efficient implementation is vital, especially in resource-constrained sensor node environment. Multithreading requires a per-thread stack, and context switching between them via scheduling principle. Hence, memory usage, energy consumption, and scheduling efficiency should be carefully considered to achieve optimal implementation. Current thread-based sensor operating systems [3, 4] do not provide specific solutions for this issue. This has motivated us to develop multithreading techniques specifically designed for micro sensor nodes

RETOS provides a few optimization techniques for the efficient implementation of multithreading on sensor node [11]. To cover the issues on memory resource, energy consumption, and thread scheduling policy, RETOS implements single kernel stack and stack-size analysis, variable timer, and event-boosting thread scheduler, respectively.

3.1 Minimizing memory usage

To reduce the memory usage for the kernel, RETOS provides two techniques: single kernel stack and stack-size analysis. Single kernel stack reduces the size of thread stack requirement, and stack-size analysis assigns an appropriate stack size to each thread automatically.

Multithread systems require stack reservation for each thread. The amount of the required stack of a thread is the sum of the resource required by thread functions, system calls, interrupt handlers and hardware context saving. RETOS implements single kernel stack management for data memory efficiency. The mechanism separates the thread stack into kernel and user stacks, and maintains a unitary kernel stack to reduce the thread stack. In the single kernel stack system, the kernel stack is shared among every thread. Controlled access to the kernel stack is implemented in such a way that the system does not arbitrarily interleave execution flow, including thread preemption, while in the kernel mode. Thread switching could be performed immediately prior to returning to user mode and executing an idle function. With thread preemption, hardware contexts are saved in each thread's thread control block due to kernel stack sharing.

With MMU-less hardware, application developers should estimate accurate thread stack size to reduce the memory usage. A stack size that is less than required by the thread causes stack overflow and easily crashes a system. Assigning a large stack would cause memory wastage. RETOS implements a stack-size analysis to provide minimal and system-safe stack for each thread, automatically. The analysis mechanism produces a control flow graph of an application. A function label, start address and internal stack usage are used as nodes in the graph, and branch instructions are used as edges. The maximum possible thread stack size is then calculated with a depth-first search. The operations are conducted on a binary image which is the result of linking application and system codes.

3.2 Reducing energy consumption with variable timer

The multithreading model of computation generally creates energy overhead due to timer management, context switching, and scheduling operation. Context switching and scheduling are known to be the source of major overhead in threaded systems. However, the frequency of scheduling in the threaded system is lower than that of passing messages between handlers in the event-driven system [12], and the context saving and restoring overhead is only a moderate issue in common sensor nodes [4].

In general-purpose threaded systems, the timer management relies on a periodic timer interrupt. This continuously triggers the

interrupt handler regardless of the timer handler request, and increases energy consumption of the sensor node which stays idle most of the time. Also, the periodic timer interrupt restricts the time accuracy within the timer interval. Instead of the periodic timer, the system may use a variable-time tick rate by way of reprogramming the tick rate with an upcoming timeout request. RETOS implements a variable timer technique to minimize the energy consumption of the multithreading system. The system timer manages timer requests from threads and updates the remaining time quantum of currently running threads. The variable timer reprograms the timer interrupt interval to the earliest upcoming timeout among the time quantum of currently running thread and the timer requests, such as the *sleep()* system-call.

3.3 Event-aware thread scheduling

RETOS supports the POSIX 1003.1b real-time scheduling interface to enable both programmers' explicit priority assignment and kernel's dynamic priority management. Threads are scheduled by three policies, SCHED_RR, SCHED_FIFO, and SCHED_OTHER [13].

SCHED_OTHER, the default scheduling policy, is specifically designed to satisfy threads used in sensor network applications with fast response time. To meet the requirement, RETOS implements an event-aware thread scheduling to increase the event response time of threads. The scheduler directly boosts the priority of the thread requesting to handle a specific event. Events in the sensor network applications are defined as the expiration of the timer request, the reception of a packet, and the completion of sensing. A thread issues a blocking system-call to handle one of these events, and the kernel enhances the thread's priority according to the type of system-call. When an event occurs, the priority-boosted thread will be able to swiftly preempt other threads. The priority of the thread reduces with the CPU time, hence other threads would have chances to be scheduled.

4. LOADABLE KERNEL MODULE

Due to the limited resources of sensor node hardware, a sensor OS can not provide all the functionality required by various types of applications at one time. Hence, a modular approach is necessary in designing the kernel. RETOS provides a mechanism to support the diverse kernel functionality that is needed only by the application at hand, enhancing kernel reconfigurability [14]. The self-reconfiguration is achieved by selecting the appropriate kernel components in the operating system without modifying the application.

4.1 Module relocation and linking

RETOS unloads unnecessary kernel modules and only maintains modules that are currently required by the applications. RETOS supports dynamic application loading and each application may require different modules. To maintain an optimized system, the kernel should always keep a minimal configuration by removing unused modules and acquiring newly-required modules, following the changes in application running.

Module implementation requires dynamic memory relocation and linking, which is not well supported by MMU-less hardware. To support a variety of microcontrollers, RETOS relies on a memory relocation mechanism, rather than a PIC (position independent code) approach. The relocation mechanism compiles the source code and extracts the relocation information of global variables and functions. The RETOS kernel acquires the compiled module with meta-information and performs code relocation. Figure 3 shows an overview of the relocation mechanism. Compiled code is stored in a RETOS file format which consists of a generic portion and a hardware-dependant section. The microcontrollers that RETOS currently supports have different addressing features, such as relocation type and relative memory-accessing instructions. Hence, we require hardware-specific information in this format to aid the relocation process for the corresponding hardware. The kernel replaces every accessible address from the code while looking up the relocation information. Since the RETOS file format has a flexible structure, any relocation mechanism in new hardware can easily be supported.

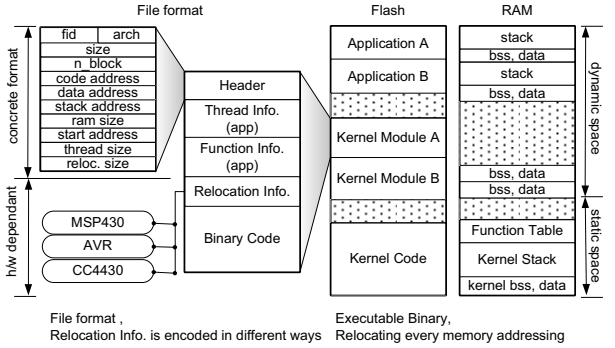


Figure 3. RETOS relocation mechanism

4.2 Module communication

RETOS maintains a function table that allows modules and applications to access other module's functions. The 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 table. The kernel and kernel modules are dynamically linked at run time and work as a single image. An application accesses the module's function through a system call which references the function table and invokes the required functions. The invoked system call performs a mode switch and validates the corresponding function. This provides protection for the kernel and kernel modules from an application's illegal memory access.

5. NETWORKING ABSTRACTION

Existing networking protocols for WSN usually adopt a cross-layering architecture; hence, an application developer sometimes takes the responsibility of developing MAC and routing protocols. This is caused by improper layering architecture in the conventional system, especially a lack of distinct layers for developing network protocols.

RETOS implements a layering architecture that provides a distinct and independent programming environment to different class of WSN developers: i.e., kernel developers, network developers, and application developers [15]. Kernel developers take responsibility for implementing core parts of operating systems so that the kernel fully utilizes the hardware. Network developers implement various networking algorithms, and using

those networking algorithms application developers program the required functions. To support this abstraction the RETOS kernel provides an appropriate protocol architecture which allows minimal modification to other layers and maximum functionality of network modules.

5.1 Overview

The RETOS network architecture provides an easy programming interface for application developers, and also enables an efficient implementation of networking functionality on a resource constrained hardware environment. The network stack consists of both static and dynamic parts. The networking functionality in the static kernel is to transmit data packets to neighbors and maintain network connectivity via neighborhood management. The static part is performance-critical, hence the implementation should be optimized at the device driver level by the kernel developer. The dynamic part of the kernel, which is normally implemented as loadable modules, implements application-dependent networking algorithms. Various kinds of routing or transport protocols can be implemented as part of the dynamic kernel.

Figure 4 shows the layered architecture of the RETOS network stack. The static kernel contains MLL and NSL. The bottom layer MLL (MAC and Data Link Layer), which controls the network devices, manages the physical connection and transmits data packets. The NSL (Networking Support Layer) supports logical connections, managing neighboring nodes and data transmission. The DNL (Dynamic Network Layer), which belongs to the dynamic part of the kernel, enables implementing network protocol algorithms and provides user API for easy development. In the RETOS network architecture, three classes of developers can implement layer-specific programs, without interfering in others, and the usage of modularization of the operating system becomes maximized. In particular, the RETOS loadable kernel module makes efficient network architecture possible. The following section describes NSL and DNL in detail.

5.2 Network Support Layer (NSL) and Dynamic Network Layer (DNL)

The NSL maintains the information of neighboring nodes and provides an interface to the DNL. The NSL consists of the Logical Link Table, Data Delivery Queue, and NSL API. The Table Manager maintains the Logical Link Table which contains the information of neighboring nodes, such as node ID, geographical position, RSSI, LQI, battery level, packet delivery ratio, packet delivery time and so on. The information collecting interval in the NSL is set periodic by network protocol developers.

In the DNL, packet routing or transport algorithms are implemented in the form of reconfigurable kernel modules. Various kinds of existing protocols can be implemented in this layer as dynamic modules. DNL modules implement core network algorithms, based on the information of neighboring nodes, network status information, and data transmission provided by the NSL API.

With the modularized protocols, the RETOS application developers build diverse networking applications by just selecting an appropriate protocol, without detailed knowledge on its implementation. To further support network transparency, RETOS provides a DNL API for application programmers, as listed below. We have considered a variety of *send* operations in order to support common communication patterns found in typical WSN applications.

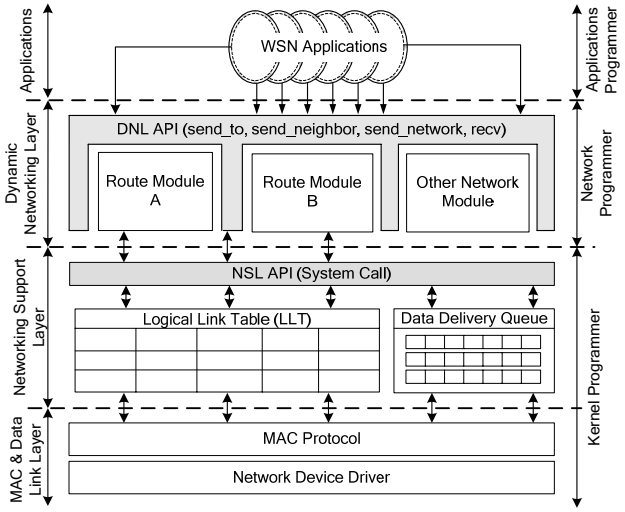


Figure 4. RETOS network architecture

route_set (): select routing module
send_to(): send to a designated node
send_neighbor(): broadcast only to neighbors
send_network(): broadcast to entire network
recv(): receive data

6. NETWORK MANAGEMENT TOOL

For any reasonable WSN programming environment, a certain form of network management tool is necessary to provide the developer with the current status of the overall network accurately and efficiently. The developer should be informed about any significant problems that need to be dealt with, such as reconfiguring the network parameters or manually relocating nodes that are not able to communicate with the network for some reason.

RETOS is accompanied by a network management system, called RMTool [16], to support component wise network management and monitoring of real sensor networks. RMTool assures a user of the network's functionality and gives the developer control over the network while running user applications over it. From the application developer's point of view, RMTool can be used as an overall view of the applications' behavior in the network, while the network health is concurrently monitored. RMTool is designed to run concurrently with other sensor network applications. As the tool should not require excessive resources over local applications, RMTool has specifically been designed to consume minimal system resources while providing a simple and robust mechanism.

7. RETOS IMPLEMENTATION

RETOS is initially implemented on the TI's MSP430-based mote, (Tmote Sky[7] and H-mote). The current version supports ATmega128 (Crossbow's MicaZ) and the CC2430 [9] SoC processor from Chipcon. Figure 5 shows the H-mote family of hardware developed in our laboratory.

(a) MPS430 H-mote (b) Sensor board (c) CC2430 H-mote

Figure 5. H-mote family of sensor nodes

One of the main objectives of RETOS is to support various types of microcontrollers. The RETOS kernel should, therefore, provide the same kernel operations on different hardware. To provide a portable kernel structure across different platforms, we have separated the architecture-independent part of the kernel from the hardware-specific part. The functionality and policies of the kernel are commonly defined in the architecture-independent part, whereas their actual implementations are optimized in the hardware-specific part depending on the individual hardware.

This section briefly explains some of the implementation issues we came across during the course of RETOS porting on different microcontrollers. The technical difficulties are mainly due to the different memory layouts, code relocation, and power management scheme of the target hardware. In particular, the memory layout and relocation mechanism provided by microcontrollers are widely different, hence we have designed an architecture-independent executable file format for the RETOS application, with which the actual code/data relocation is conducted in binary load stage. The detailed relocation mechanism is implemented architecture-dependent way. Figure 6 shows the overview of RETOS code layout on various architectures. The power management mechanism also varies with the microcontrollers. In RETOS, basic power management is conducted in an architecture-independent part, whereas the detailed power control is specifically carried out by the underlying processor.

Table 1 shows the code size comparison of fully-equipped RETOS kernel and TinyOS v1.1.15, where various sensor drivers as well as network module are included. The case of minimal configuration of TinyOS without any sensor drivers or network module is also given in the table for the reference.

7.1 TI MSP430

Different from the AVR or CC2430 memory layout, MSP430 enables the access to the entire flash area without memory bank or page selection. Hence, the kernel code section is implemented as it is in the RETOS code image layout, as illustrated in Figure 6. The compiled kernel image is stored in the starting address of flash. The remaining flash area is used for application loading or dynamic kernel module. In the case of MSP430 F1611 which has 48KB of flash memory, approximately 24KB of flash can be used by application or run-time modules.

The relocation mechanism of MSP430 can optimally be implemented by using a single-bit of information allocated for each word. In MSP430, knowing the start addresses of code and data sections, the relocation is made possible by checking the address range of the operands in an instruction if it is a code or data. Any instruction with the single-bit set is relocated. The bit-based relocation mechanism in MSP430 has a storage overhead of 1/16 of the code size.

MSP430 provides five levels of power operating modes: LPM0 to LPM4. For idle running, RETOS runs in LPM1 which halts the processor operation and the system clock. Also, depending on which sensor uses the digitally-controlled oscillator, the processor runs in LPM3 to further reduce the energy consumption.

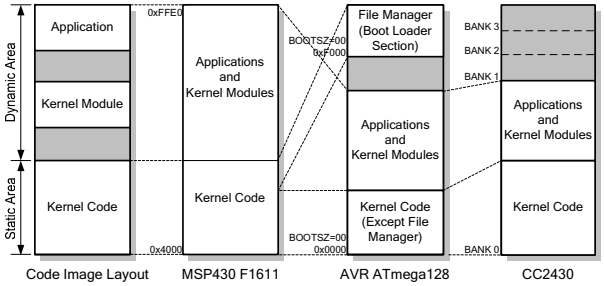


Figure 6. RETOS code layout on various architecture

Table 1. Code size comparison

	MSP430		AVR		CC2430	
	ROM	RAM	ROM	RAM	ROM	RAM
RETOS Kernel	23132	824	24202	868	28720	820
TinyOS v1.1.15	20924	798	17160	792	N/A	
TinyOS (Min.)	11286	311	9500	332	N/A	

7.2 Atmel ATmega128

ATmega128 addresses the SRAM area and the 128KB of internal flash separately. Also, the internal flash is divided into a boot flash section and an application flash section. Any operation for data read or write on internal flash should be executed in the boot flash section starting from $0xf000, hence a part of the RETOS kernel resides in the boot area while the rest is loaded into the application flash area. The application loading on ATmega128 can be easily implemented with the PIC approach, but the mechanism has a limitation when addressing large amount of code, and more importantly, many microcontrollers do not support this mechanism. RETOS implements the address relocation mechanism, instead. Since ATmega128 is an 8-bit microcontroller with a 16-bit addressing mode, the relocation mechanism is complex, compared to the case of MSP430. ATmega128 has 19 different types of relocation, depending on the instruction type as well as the addressing range of the operand. The relocation information field in the executable file format in Figure X enables the relocation of individual applications. For each relocating instruction, five bytes are used for the field: two bytes for instruction offset, one byte for relocation type, and another two bytes for address.

Six low-power modes are provided by ATmeg128. RETOS makes use only of the “standby” mode when the processor becomes idle, but more sophisticated power management is currently under development.

7.3 Chipcon CC2430

CC2430 is the latest SoC microcontroller from Chipcon, which integrates the 8051-based processor core running at 32MHz and the IEEE 802.15.4-compatible RF module. Due to its small form factor and high performance, CC2430 is considered a useful

building block for WSN hardware platforms. Out of 128Kbytes of flash memory available in CC2430, only 55KB can be used for program memory, where the bottom 32KB should be used for the boot and kernel codes, and the remaining 23KB are used for application and libraries. No relocation is required for the bottom 32KB, but the upper 23KB needs extra operations, due to the different flash bank, and the codes are mapped to XDATA space and the relocation is conducted appropriately. The network performance of RETOS is enhanced by using the RADIO DMA trigger, which enables efficient and power-aware data processing by not waking up the processor. The MAC timer provided by CC2430 also aids the efficient execution of the CSMA/CA protocol.

CC2430 provides four levels of power operating mode: PM0 to PM3. PM0 is a full-functioning mode, PM1 and PM2 the low-power modes, and PM3 the lowest power mode without running the clock. Based on the variable-timer operation, RETOS makes use of PM1 and PM2 modes aggressively, depending on the current hardware usage on sensor I/O or network I/O. While operating in PM1 or PM2 modes, the sleep timer, which runs on a 32.768KHz oscillator, is used to support interrupt handling and to maintain the system timer.

8. EVALUATION

Evaluating an operating system is not an easy task, and requires a manifold and sophisticated methodology. Each OS component, as well as their integrated operations, can be analyzed separately, or evaluated comparatively with other operating systems. We have previously evaluated the performance of RETOS [10,11,14,15,16], along with various design issues, and interested readers may find the quantitative results published in our earlier work useful. In this paper, we evaluate RETOS from the perspective of overall system operation, rather than individual component-wide characteristics, while running some of the real-world WSN applications. We demonstrate how the design principles of RETOS work with real applications.

Three applications are used for the evaluation. First, a multiple-object tracking system (MPT) is implemented with RETOS in an indoor environment, based on ultrasonic sensor devices. MPT is a good example of evaluating the efficiency of thread-based programming. Second, we have implemented a large-scale node localization mechanism using mobile acoustic sources (ASL), based on inexpensive MICs, in RETOS. With this application, the concurrency feature of RETOS is evaluated, together with the usefulness of the network management tool. Finally, a distributed acoustic source detection system (DSLS) is implemented with RETOS. The system enables the evaluation of effectiveness of RETOS modules and network architectures. Table 2 shows the code size analysis of the RETOS applications running on MSP430, compared to TinyOS.

Table 2. Code size for applications (MSP430)

ROM/RAM	TinyOS	RETOS				
MPT	Total	kernel	apps	routing*†	-	-
mobile	20680 / 677	23132 / 824	7710 / 154	N/A	-	-
backbone	20200 / 1170	23132 / 824	724 / 205	2358 / 356	-	-
ASL	Total	kernel	time sync	ASL	RM tool	-
	26674 / 994	23132 / 824	2136 / 200	8928 / 286	2278 / 146	-
DSLS	Total	kernel	time sync*	DSLS	RM tool*	routing*†
	30358 / 1636	23132 / 824	2168 / 254	6698 / 244	2298 / 146	2358 / 356

* kernel module implementation † parametric routing [21]

8.1 Multiple object tracking

Ultrasound-based object tracking is a common indoor WSN application, due to its low-cost ad high-accuracy advantages. In our previous work [17], we developed an active tracking system for multiple moving objects (MPT) in a TinyOS environment. Our hardware platform is compatible to the Telos mote, but equipped with an ultrasonic sensor module that detects 40KHz ultrasonic pulses. With the availability of RETOS, we have ported MPT to the same hardware. This section describes the implementation and evaluation of the system.

MPT consists of mobile nodes and backbone nodes. The mobile nodes periodically send synchronized radio signals and ultrasound pulses. Upon receiving a radio signal, the backbone node measures the arrival time difference between the radio signal and the ultrasound pulse, calculates the distance, then sends it back to the mobile node. With three pieces of distance information calculated from backbone nodes, the mobile node estimates its location in the 2D plane coordinate by a trilateration algorithm. This process is continued in every beaconing period of the mobile nodes, enabling the tracking of mobile nodes. To solve multiple mobile nodes transmitting the beacon signals in an overlapped fashion, an adaptive beaconing algorithm is implemented, with which mobile nodes overhear the beacon messages of other mobile nodes and adaptively adjust their beaconing periods to avoid a simultaneous ultrasound pulse.

Compared to the TinyOS implementation of MPT, a simple and intuitive implementation was possible with RETOS. Figure 7, for example, outlines the program structures, written in TinyOS and RETOS, running on the MPT backbone node. The control flow is straightforward, but the TinyOS implementation requires a careful split of tasks and proper sequencing of the event handlers. With RETOS, an application programmer can implement the code sequentially in a single thread. A set of API functions for the sensor device, provided by RETOS, also eases the implementation effort.

Apart from the programming convenience of RETOS, we now discuss the efficiency of the threaded system. The MPT mobile node executes trilateration after receiving distance information from three beacon nodes. The trilateration is a CPU-bound job and takes approximately 16ms on MSP430. Each beaconing should be sufficiently separated by *Twait*, as the backbone node needs processing time. Figure 8 compares the execution model of the mobile node for TinyOS and RETOS. With TinyOS, the node cannot handle events during the trilateration, nor overhear the beacon messages from other mobile nodes. With RETOS, however, a thread is created only for trilateration, hence the main thread can concurrently handle the beacon messages. Our experiments show that the MPT system running on MSP430-based motes with TinyOS handles four mobile nodes at most with 300ms of beacon period. By reducing the beaconing period by 16ms, RETOS enables a smoother operation of object tracking, or alternatively, RETOS can support up to five mobile nodes with the original beacon period.

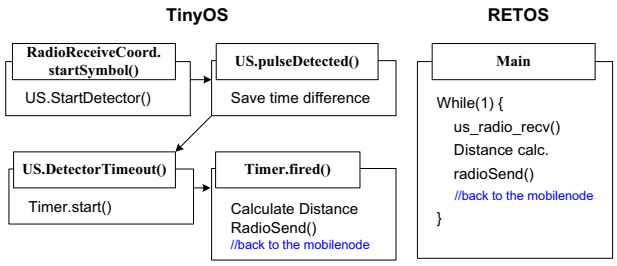


Figure 7. Program structures of MPT backbone

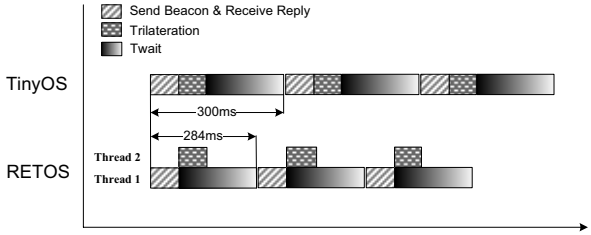


Figure 8. Execution model for MPT mobile

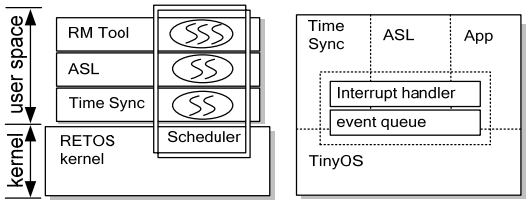
8.2 Localization using mobile acoustic sources

Sensor node localization is a fundamental requirement for large-scale and practical use of WSN applications. We previously developed a large-scale localization system (ASL) using mobile acoustic sources, and implemented it in a TinyOS environment [18]. We have recently ported ASL with RETOS.

ASL consists of two types of nodes: Sound generator (SG) and Sound Detector (SD). SG generates an acoustic event, while the same event will be detected by SDs. With the time synchronization among all nodes, an acoustic event generated near a SG node is first detected by its own microphone sensor. The time of detection at the sensor is used as the time of event generation, and is sent to all SD nodes. SD nodes will eventually detect the acoustic event, confirm the time of event generation with data sent by the SG node, and store the time difference and beacon coordinates within a specific range of acoustic events. Repeating this process, the SD nodes continue to collect acoustic event coordinates. With three coordinates gathered, all SD nodes use the distance data and beacon coordinates to independently localize its own position using trilateration.

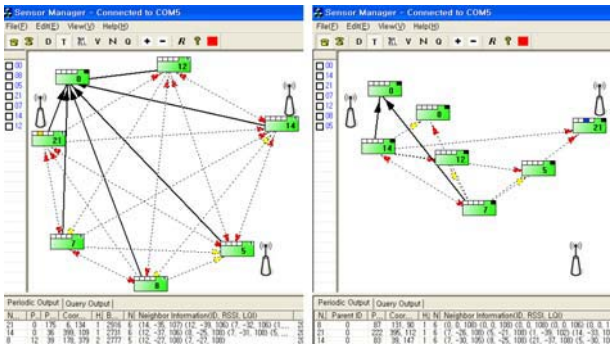
To run the application we need a global time synchronization module running on each node. We have implemented FTSP [19] as a multithreaded application on RETOS. The network management tool RMTool mentioned in Section 6 is also running on the node, as an application, to monitor the progress of node localization as well as the network status. Overall, the SD node has three applications running seven threads concurrently (2 for ASL, 2 for FTSP, and 3 for RMTool). Figure 9 illustrates the program structures of ASL in RETOS and TinyOS environment. The experiment results, with five MSP430-based motes deployed in outdoor environment, showed that three separate applications run accurately and efficiently in the RETOS environment, and the overall ASL performance is as good as in our TinyOS implementation. Figure 10 shows the RMTool screenshots for ASL, before and after the execution.

With this experiment, we purposely pushed the capability of the RETOS execution environment, in terms of resource usage and performance, by running three non-trivial applications at the same time. We are quite satisfied with the overall performance of the system. Although ASL, FTSP and RMTool were run as "applications" in this setup, these are in fact core components of the WSN operating environment. Hence, modularizing them as kernel modules is an alternative approach to optimize the kernel, because each component can be selectively loaded in the system depending on the application at hand.



(a) RETOS (b) TinyOS

Figure 9. ASL implementation on RETOS and TinyOS



(a) Before localization (b) After localization

Figure 10. Screenshots of RMTool for ASL

8.3 Acoustic source detection

Acoustic source localization, which automatically detects the location of a sound source is a good example of WSN and the technique can be applied to many real-world applications. Previously, we developed a distributed, low-complexity, range-free, and error-tolerant acoustic source localization system (DSLS) in the TinyOS environment [20]. We have recently ported DSLS with RETOS, and this section details the implementation.

Upon an acoustic event in DSLS, a group, which is a unit of acoustic source localization, is constructed on the fly with all the listening nodes of the event. By comparing the listening times of all nodes in the group, a leader is selected as being the closest to the acoustic source. A voting grid is then constructed in the leader node. The voting grid is a two-dimensional array of fields which is used to estimate the location of the acoustic source. Every node in the group votes for the possible location of the acoustic event separately, and the voting results are collected in the leader node to finalize the location.

In the RETOS implementation of DSLS, we were particularly interested in the usage of DNL API provided by the RETOS

network stack. Evaluating the usage and practicality of the RETOS module was another goal of this experiment. A RETOS programmer is provided with several communication primitives. In the case of DSLS, the *send_neighbor()* function is effectively used to elect a leader and to exchange the voting results in a single-hop neighbor group. The final result is delivered to the multi-hop away sink in an end-to-end fashion using the *send_to()* primitive, which is implemented by a specific routing algorithm as a loadable kernel module in RETOS. For DSLS, we implemented the parametric routing [21] as a kernel module. The parametric routing considers energy, reliability, and timeliness issues together in a single framework to select an appropriate next-hop node upon user's performance preference.

In addition to the routing module, we re-implemented FTSP and RMTool as kernel modules for the use with DSLS in order to evaluate the feasibility of running core WSN elements as loadable module in RETOS. For the experiment, ten MSP430-based motes with MICs were deployed in 20m*20m outdoor to detect sound sources such as hand clapping or wood sticks being struck together. The overall system was working as correct as in the TinyOS environment and the location error was less than 1 meter for most of the cases. The loadable module of RETOS has proven useful and practical in a way that we can dynamically optimize the kernel configuration.

9. RELATED WORK

Current operating systems for WSN include TinyOS [1], SOS [2], Contiki [3], MANTIS [4], and t-kernel [5]. They are broadly classified into two programming models: event-driven and multithreading. TinyOS [1] is based on the event-driven model and nesC [22] is used to program the application and system software. TinyOS produces a single code image where the kernel and application are statically linked. This feature enables compile-time optimization with a function inlining. However, the overhead for updating the entire kernel code is not trivial. In TinyOS, the kernel is not protected from the application, hence a badly-written application may cause the system to fail. A watchdog timer can be used for this purpose, but recognizing and handling the application errors are not easy. So far, TinyOS has been implemented on AVR and MSP430 family of microcontrollers. Based on event-driven model, SOS [2] provides dynamically loadable modules. The modular approach, which separates the application from the system, enables the easy modification of the functions for the applications' needs. Recently, Kumar et al. [23] implemented software-based fault isolation [24] on SOS, although the technique only protects the kernel data memory.

Contiki [3], which is also an event-driven system, has implemented loadable modules using relocation and CELF [25]. The PIC technique, used by SOS, is not applicable to many microcontrollers, but Contiki somehow overcomes the portability issues of loadable modules. The Contiki system does not provide a safety mechanism for abnormal behavior of the kernel or application. TinyOS, SOS, and Contiki all adopt the event-driven model to minimize the overhead of multithreading. With event-based systems, however, programmers must split long-lived tasks into several phases of codes for concurrency, and construct explicit machine states manually. Protothread [26] is proposed to use blocking functions on top of event-driven systems without stack reservation; however the mechanism is unable to maintain local variables and blocks only in an explicitly declared area.

MANTIS [4] is probably the first sensor OS to support the multithreaded programming environment. MANTIS showed that programming long-running tasks is easier with multithreading than with event-driven model. However, the current implementation has some limitations. For example, the programmers heuristically assign stack size to each thread and adjust a thread priority manually. The MANTIS scheduler, based on fixed-interval timer interrupt, could possibly delay response time for threads. The context switching and timer interrupt handling in MANTIS is not optimized. MANTIS produces a statically-linked single code image, hence reprogramming cost is not trivial, as in TinyOS, and the kernel or application may crash due to the lack of protection mechanism.

t-kernel [5] provides a software-based memory protection and virtual memory via load-time code modification. The system necessarily expands code size and incurs run-time overhead, because every memory access or code branch requires address transition or memory swapping. Gu et al. [5] briefly mentioned that t-kernel provides preemptive scheduling, but the detailed execution model is unclear in the literature.

Kernel supports for WSN networking have been studied in terms of network abstraction. Ye [27] implemented WSN protocols in the conventional MAC layer framework. Dunkels [28] made an effort to implement the TCP/IP stack in 8-bit AVR MCU, Kumar [6] suggested a data-centric network stack, and related network layering for data fusion. SP (sensornet protocol) [29] has recently been suggested as a translucent and unifying link abstraction for WSN.

Active development is also given to provide useful network management tools for WSN. Among them are Mote-View[30], SNMS [31], TinyCubus [32], and Sympathy [33]. These tools aid the configuration, monitoring, and management of the deployed sensor nodes.

10. CONCLUSIONS

Although active research and development is currently being conducted in wireless sensor network communities, it is fair to say that the practitioners in the field do not have many choices regarding the operating system and related programming development tools. From the very early days of WSN research, the event-driven TinyOS has been considered the industry de-facto operating system, due to its stability and efficiency; hence, the software developers should understand the OS principle and accordingly stick to its programming model for better outcomes. Having experienced TinyOS ourselves, we have a different view on the "right" form of WSN operating system. Our belief is that a sensor OS should provide an easy programming interface, both for WSN experts and general application programmers, by concealing the underlying OS principles from users. In this context, we believe that a multithreaded OS fits this criteria better. Our other motivation is that a sensor OS should be robust and resilient in a sense that the kernel or applications should not crash unexpectedly even on MMU-less hardware. System expandability based on a reconfigurable kernel is also thought to be an essential OS feature, as well as the provisioning of the WSN-specific networking abstraction in the kernel. With this paper, we are not arguing that RETOS is superior to existing sensor OS, but that an

alternative approach for sensor OS design is indeed feasible and practical, and further enables a wider choice of sensor OS for application developers or system programmers.

RETOS is not just an experimental OS for research purposes, but a fully functional and extensively tested operating system with non-trivial, real-life applications, as discussed in Section 8. We have been using RETOS in undergraduate and graduate courses here in Yonsei University as a teaching and research platform for the last couple of semesters. The operating system is fairly stable at the moment, although the performance is being tuned with added OS functionality.

We are presently analyzing the performance characteristics of the newly-ported RETOS on AVR-based MicaZ and CC2430-based H-mote. The results will hopefully be published soon. Other efforts include the development of the GUI-based RETOS programming environment (IDE), remote debugging tool, and more RETOS porting to other microcontrollers.

11. ACKNOWLEDGMENTS

This research was supported by the National Research Laboratory (NRL) program of the Korean Science and Engineering Foundation (2006-01546) and the MIC (Ministry of Information and Communication)'s ITRC program (IITA-2006-C1090-0603-0015)

12. REFERENCES

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, Kristofer Pister, "System architecture directions for network sensors," *In Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS),* Cambridge, MA, November 2000.

[2] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, M. Srivastava, "SOS: A dynamic operating system for sensor networks," *In Proc. of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys),* Seattle, WA, June 2005.

[3] A. Dunkels, B. Grönvall, T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," *In Proc. of the First IEEE Workshop on Embedded Networked Sensors (EmNets)*, Tampa, Florida, November 2004.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms," *ACM/Kluwer Mobile Networks & Applications, Special Issue on Wireless Sensor Networks,* vol. 10, no. 4, August 2005.

[5] L. Gu, J. A. Stankovic, "t-kernel: Providing Reliable OS Support to Wireless Sensor Networks," *In Proc. of the 4th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Boulder, Colorado, 2006.

[6] R. Kumar, S. PalChaudhuri, D. Johnson, U. Ramachandran, "Network Stack Architecture for Future Sensors," Rice University, Computer Science, Technical Report, TR04-447.

[7] Tmote Sky, http://www.moteiv.com

[8] MicaZ, http://www.xbow.com

[9] CC2430, http://www.chipcon.com

[10] H. Kim, H. Cha, "Towards a Resilient Operating System for Wireless Sensor Networks", *In Proc. of the 2006 USENIX Annual Technical Conference*, Boston, Massachusetts, June 2006.

[11] H. Kim, H. Cha, "Multithreading Optimization Techniques for Sensor Network Operating Systems," In Proc. of the 4th European conference on Wireless Sensor Networks (EWSN), Delft, Netherlands, January 2007.

[12] R. Behren, J. Condit, E. Brewer, "Why events are a bad idea (for high-concurrency servers)," *In Proc. of the 9th Workshop on Hot Topics in Operating Systems (HotOS)*, Lihue, Hawaii, 2003.

[13] POSIX 1003.1B, http://www.unix.org/version3

[14] H. Shin, H. Cha, "Supporting Application-Oriented Kernel Functionality for Resource Constrained Wireless Sensor Nodes," *In Proc. of the 2nd International Conference on Mobile Ad-hoc and Sensor Networks (MSN 2006),* Hong Kong, China, December 2006.

[15] S. Choi, H. Cha, "Application-Centric Networking Framework for Wireless Sensor Nodes," *In Proc. of the 3rd Annual International Conference on Mobile and Ubiquitous Systems (MOBIQUITOUS)*, San Jose, California, July 2006.

[16] I. Jung, H. Cha, "RMTool: Component-Based Network Management System for Wireless Sensor Networks," *In Proc. of the 2007 IEEE Consumer Communications and Networking Conference (CCNC),* Las Vegas, January 2007.

[17] S. Yi, H. Cha, "Active Tracking System using IEEE 802.15.4-based Ultrasonic Sensor Devices," *In Proc. of the 2nd International Workshop on RFID and Ubiquitous Sensor Networks (USN)*, Seoul, Korea, August 2006.

[18] Y. Lee, H. Cha, "A Light-weight and Scalable Localization Technique Using Mobile Acoustic Source," *In Proc. of the 2006 IEEE International Conference on Computer and Information Technology (CIT 2006)*, Seoul, Korea, September 2006.

[19] M. Maróti, B. Kusy, G. Simon, A. Ledeczi, "The Flooding Time Synchronization Protocol," *In Proc. of the 2nd ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004.

[20] Y. You, H. Cha, "Scalable and Low-Cost Acoustic Source Localization for Wireless Sensor Networks," *In Proc. of the 3rd International Conference on Ubiquitous Intelligence and Computing (UIC)*, Wuhan and Three Gorges, China, September 2006.

[21] Y. Sung, H. Cha, "Parametric Routing for Wireless Sensor Networks," *In Proc. of the 2006 International Symposium on Ubiquitous Computing Systems (UCS)*, Seoul, Korea, October 2006.

[22] D. Gay, P. Levis, R. Behren, M. Welsh, E. Brewer, D. Culler, "The nesC Language: A Holistic Approach to Network Embedded Systems," *In Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2003.

[23] R. Kumar, E. Kohler, M. Srivastava, "Software-Based Memory Protection In Sensor Nodes," *In Proc. of the Third Workshop on Embedded Networked Sensors (EmNets)*, Cambridge, MA, 2006.

[24] R. Wahbe, S. Lucco, T. E. Anderson, S. L. Graham, "Software-based fault isolation," *In Proc. of the 14th ACM Symposium on Operating System Principles (SOSP)*, Asheville, NC, USA, December 1993.

[25] A. Dunkels, N. Finne, J. Eriksson, T. Voigt, "Run-time Dynamic Linking for Reprogramming Wireless Sensor Networks," *In Proc. of the 4th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Boulder, Colorado, November 2006.

[26] A. Dunkels, O. Schmidt, T. Voigt, "Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems," *In Proc. of the 4th ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Boulder, Colorado, November 2006.

[27] W. Ye, J. Heidemann, D. Estrin, "*A Flexible and Reliable Radio Communication Stack on Motes*," USC/ISI Technical Report ISI-TR-565.

[28] A. Dunkels, "Full TCP/IP for 8 Bit Architectures," *In Proc. of the 1st ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys)*, San Francisco, May 2003.

[29] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, I. Stoica, "A Unifying Link Abstraction for Wireless Sensor Networks," *In Proc. of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, San Diego, November 2005.

[30] Mote-View, http://www.xbow.com.

[31] G. Tolle, D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," *In Proc. of the 2nd European Workshop on Wireless Sensor Networks (EWSN),* Istanbul, Turkey, January 2005.

[32] P. J. Marr´on, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, K. Rothermel, "TinyCubus: A Flexible and Adaptive Framework for Sensor Networks," *In Proc. of the 2nd European Workshop on Wireless Sensor Networks,* Istanbul, Turkey, January 2005.

[33] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, D. Estrin, "Sympathy for the Sensor Network Debugger," *In Proc. of the 3rd international conference on Embedded networked sensor systems (Sensys),* San Diego, CA, 2005.