

# MANTIS: System Support for Multimodal Networks of In-situ Sensors

H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, R. Han  
University of Colorado at Boulder  
Department of Computer Science  
Contact author: rhan@cs.colorado.edu

## ABSTRACT

The *MANTIS* Multimodal system for Networks of In-situ wireless Sensors provides a new multithreaded embedded operating system integrated with a general-purpose single-board hardware platform to enable flexible and rapid prototyping of wireless sensor networks. The key design goals of MANTIS are ease of use, i.e. a small learning curve that encourages novice programmers to rapidly prototype novel sensor networking applications in software and hardware, as well as flexibility, so that expert researchers can leverage or develop advanced software features and hardware extensions to suit the needs of advanced research in wireless sensor networks.

## Categories and Subject Descriptors

D.4.7 [Operating Systems]: Organization and Design - *real-time systems and embedded systems, interactive systems*. C.3 [Computer Systems Organization] Special-purpose and Application-based Systems - *real-time and embedded systems*.

## General Terms

Design, Experimentation, Performance, Security, Human Factors.

## Keywords

Wireless sensor networks, operating systems, lightweight, multimodal prototyping, dynamic reprogramming, GPS.

## 1. INTRODUCTION

The growing popularity of wireless sensor networks (WSNs) has placed increasing demands upon the infrastructure of today's general-purpose hardware/software sensor systems [1,2,3,4] to support improved flexibility, ease of use, and lower cost. The *MANTIS* Multimodal system for Networks of In-situ wireless Sensors provides a new multithreaded embedded operating system integrated with a general-purpose single-board hardware platform to enable flexible and rapid prototyping of wireless sensor networks. The key design goals of MANTIS are ease of use, i.e. a small learning curve that encourages novice

programmers to rapidly prototype novel sensor networking applications, as well as flexibility, so that expert researchers can continue to adapt and extend the hardware/software system to suit the needs of advanced research.

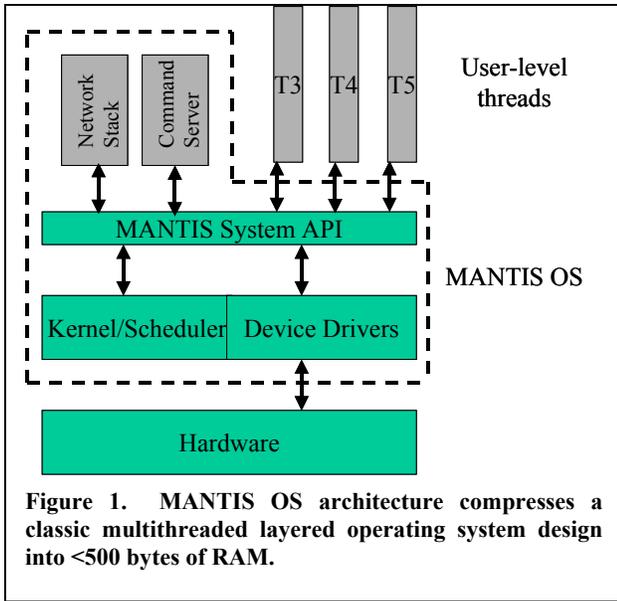
The first goal in the design of the MANTIS OS (MOS) was to meet the objective of ease of use or convenience while also adapting MOS to the resource constraints of wireless sensor networks, namely limited memory and power. To lower the barrier to entry into the field of sensor networks and encourage novice application developers, early design choices of MOS adhered to familiar themes in programming languages and operating systems design. For these reasons, MOS selected as its model the classical structure of layered multithreaded operating systems, which includes multithreading, pre-emptive scheduling with time slicing, I/O synchronization via mutual exclusion, a standard network stack, and device drivers. Familiarity with these classical structures lowers the learning curve for novice developers. A key challenge has been to adapt these classical structures to the limited memory of sensor nodes. At present, the MOS kernel is able to achieve multithreaded pre-emptively scheduled execution with standard I/O synchronization and a network protocol stack, all for less than 500 bytes of RAM, not including individual thread stack sizes.

Another means by which MOS achieves the goal of ease of use is via its choice of a standard programming language. In particular, the entire kernel and API are written in standard C. This design choice not only considerably flattens the learning curve, due to the vast number of programmers with prior experience in C, but also accrues many of the other benefits of a standard programming language, including cross-platform support and reuse of a vast legacy code base. For example, a standard stop-and-wait reliable protocol as well as a standard RC5 security algorithm [5] are both available in C, and have been ported into the MOS kernel. The choice of C also eases development of cross-platform multimodal prototyping environments on X86 PCs, as explained below. As a result, MOS has the potential to considerably shorten development cycles by enabling rapid prototyping of applications as well as rapid testing and debugging of additions and modifications to our MOS kernel.

The second objective of MOS is to promote flexibility for advanced research in sensor networks. Towards this end, MOS supports useful yet sophisticated features that are tailored to advanced sensor networks, including dynamic reprogramming of sensor nodes via wireless, remote debugging of sensor nodes, and multimodal prototyping of virtual and deployed sensor nodes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WSNA '03, September 19, 2003, San Diego, California, USA.  
Copyright 2003 ACM 1-58113-764-8/03/0009...\$5.00.



In the remainder of the paper, Section 2 describes the MOS architecture, how it offers a convenient environment for development of WSN applications., and how it achieves a lightweight implementation. Section 3 provides a detailed overview of advanced MOS features. Section 4 focuses on the hardware. Section 5 describes deployed applications, and Section 6 finishes with future work.

## 2. LIGHTWEIGHT MANTIS OPERATING SYSTEM DESIGN

In this section, we describe the architecture of the MANTIS operating system, which adheres to a classical layered multithreaded design, as shown in Figure 1. The top application and API layers provide an opportunity to observe how the choice of a simple C API promotes ease of use, cross-platform portability, and reuse of a large installed code base. In the lower layers of MOS, we describe our novel adaptation of classical OS structures to achieve a small memory footprint.

### 2.1 Applications

MANTIS provides a convenient environment for creating WSN applications. Figure 2 illustrates a simple yet commonly used “sense\_and\_forward” application, which is available along with the complete MANTIS software release 0.1 at <http://mantis.cs.colorado.edu/>. This simple application, which runs on a sensor Nymph (see Section 4), toggles a yellow LED, reads a sensor value from an analog to digital converter (ADC) port and then transmits the value of the sensor over the radio.

All applications begin with **start**. Before transmitting, the network must be initialized first by calling **net\_init** before any threads are spawned. Then two threads are spawned with **thread\_new**. One call spawns the **mos\_inetd**, which is provided, and the other call spawns the **test\_adc\_send** application, which has been developed by the user. Thus, an application may conveniently be built of more than one thread.

### sense\_and\_forward.c

```
#include <stdio.h>
#include "led.h"
#include "scheduler.h"
#include "network.h"
#include "adc.h"

void test_adc_send();

void start(void){
    net_init(CHANNEL_1);
    thread_new(mos_inetd, 128,
    PRIORITY_NORMAL);
    thread_new(test_adc_send, 128,
    PRIORITY_NORMAL);
}

void test_adc_send(){
    uint8_t value;
    adc_open();
    set_addr(0x11);
    set_radio_power(0xff);

    while(1){
        led_yellow_toggle();
        value =
    adc_convert_eight_polling(ADC_CH_2);
        mos_send_to(0x41, 0x02, &value,
    0x01, FLOODING);
    }
}
```

**Figure 2. Sample application C code, sense-and-forward.c.**

Within the **test\_adc\_send** thread, reading a sensor requires that an analog-to-digital converter (ADC) be opened with **adc\_open**. The node address is set as well as the radio frequency power, by calling **set\_addr** and **set\_radio\_power**, respectively. With these preliminaries completed, the application can now toggle an LED, read the sensor value from the ADC and send it over the radio, by calling the **led\_yellow\_toggle**, **adc\_convert\_eight\_polling** and **mos\_send\_to**.

All of the function calls mentioned so far are part of the MANTIS System Application Programming Interface (API). The program is compact and requires a fairly shallow learning curve for C programmers. Early empirical experience with MOS suggests that application developers can rapidly prototype new applications in this environment. Applications such as a sensor-enabled conductor's wand [6] of Section 5 were prototyped in hours, while applications such as a frequency-hopping protocol and a port of the RC5 security standard were completed in less than two nights.

### 2.2 System APIs

MANTIS provides a comprehensive set of System APIs for I/O and system interaction. For a complete list and information on all the APIs please refer to <http://mantis.cs.colorado.edu/>. For the

preceding **sense\_and\_forward** application example, the APIs that were used in the application can be categorized as:

Networking: <b>net_init, set_addr, set_radio_power, mos_send_to</b>
On board sensors (ADC): <b>adc_convert_eight_polling, adc_open</b>
Visual Feedback (LEDs): <b>led_yellow_toggle</b>
Scheduler: <b>thread_new</b>

The choice of a C language API simplifies cross-platform support and the development of a multimodal prototyping environment. The MANTIS System API is preserved across both physical sensor nodes as well as virtual sensor nodes running on X86 platforms. As a result, the same C code developed for MANTIS sensor Nymphs with ATMEL microcontrollers [7] can be compiled to run on X86 PCs with little to no alteration.

### 2.3 Kernel and Scheduler

The design of the MOS kernel resembles classical, UNIX-style schedulers. The services provided are a subset of POSIX threads [8], most notably priority-based thread scheduling with round-robin semantics within a priority level. Binary (mutex) and counting semaphores are also supported. The goal of the MOS kernel design is to implement these familiar services in a manner efficient enough for the resource-constrained environment of a sensor node.

The most limited resource on a MANTIS node is the RAM. There are two logically distinct sections of RAM: the space for global variables that is allocated at compile time, and the rest of RAM that is managed as a heap. When a thread is created, stack space is allocated by the kernel out of the heap. The space is recovered when the thread exits. In the current implementation, the user is not able to dynamically allocate heap space, although that was an API decision and is not an inherent limitation of MOS.

The kernel's main global data structure is a thread table, with one entry per thread. Since the thread table is allocated statically, there is a fixed maximum number of threads and a fixed level of memory overhead. The maximum thread count is adjustable at compile time (the default is 12). Each thread table entry is ten bytes and contains a current stack pointer, stack boundary information (base pointer and size), a pointer to the thread's starting function, the thread's priority level, and a next thread pointer for use in linked lists. Note that pointers on the AVR microcontroller are only two bytes. A thread's current context, including saved register values, is stored on its stack when the thread is suspended. This is significant, because the context is much larger than a thread table entry, and it only needs to be stored when the thread is allocated. Thus the static overhead of the thread table is only 120 bytes.

The kernel also maintains ready-list head and tail pointers for each priority level (5 by default, for 20 bytes total). Keeping both pointers allows for fast addition and deletion, which improves performance when manipulating thread lists. This is important because those manipulations are frequent and always occur with

interrupts disabled. There is also a current thread pointer (2 bytes), an interrupt status byte, and one byte of flags. The total static overhead for the kernel is thus 144 bytes.

Semaphores in MOS are 5-byte structures that are declared as needed by applications; they contain a lock or count byte along with head and tail list pointers. At any given time, each allocated thread is a member of exactly one list; either one of the ready lists or a semaphore list. Semaphore operations move thread pointers between lists, and the scheduler cycles through the ready lists to locate the next thread to execute.

The scheduler receives a timer interrupt from the hardware to trigger context switches; switches may also be triggered by system calls or semaphore operations. The timer interrupt is the only one handled by the kernel--other hardware interrupts are sent directly to the associated device drivers. Upon an interrupt, a device driver typically posts a semaphore in order to activate a waiting thread, and this thread handles whatever event caused the interrupt. There are currently no 'soft' interrupts supported by the MOS kernel, although the design does not preclude adding them in the future.

In addition to driver threads and user threads, there is also an idle thread created by the kernel at startup. The idle thread has low priority and runs when all other threads are blocked. The idle thread is in a position to implement power-aware scheduling, as it may detect patterns in CPU utilization and adjust kernel parameters to conserve energy.

### 2.4 Network Stack

Wireless networking is critical for the correct operation of a network of sensors. Such communication is typically realized as a layered network stack, not to be confused with the thread stack. The design of the MANTIS network stack is focused on efficient use of limited memory, flexibility, and convenience. The stack is implemented as one or more user-level threads. Different layers can be flexibly implemented in different threads, or all layers in the stack can be implemented in one thread. The tradeoff is between performance and flexibility. The stack is designed to minimize memory buffer allocation through layers. The data body for a packet is common through all layers *within* a thread. The headers for a packet are variably-sized and are pre-pended to the single data body. The stack is conveniently designed in a modular manner, with standard APIs between each layer, thereby allowing developers to easily modify or replace layer modules. The routing protocol is assigned on a per packet basis, so that different routing mechanisms can coexist, including flooding, multicast, and unicast. The stack flexibly supports multi-frequency radio communication over 30 channels, enabling research into MAC protocol design, security and reliability. A flexible range of packet sizes is supported, from 12 bytes to 64 bytes, avoiding waste of scarce sensor network bandwidth.

The network stack consists of four layers, i.e. application layer, network layer, MAC layer and physical layer. At the bottom is the physical layer implementation that controls the hardware or virtual hardware (for XMOS). A set of standard APIs is provided on top of the physical implementation in order to mask the underlying hardware details from the MAC Layer. The MAC layer is responsible for controlling such aspects as network duty cycle, wherein the radio is adaptively slept to save on energy consumption.

Together, the MAC and physical layers are realized as one user-level thread, as shown in Figure 1, and is known as the base thread of the network stack. MANTIS extends the concept of user-level implementation of network stacks from such projects as ALPINE [9] to the sensor networking domain, and outlines the advantages of this approach below. The base network stack thread blocks on a well-known semaphore. When a packet arrives at the radio interface, the interrupt is handled by a device driver, which places the data in a queue and then posts the semaphore where the base network stack thread is sleeping. This wakes the base network stack thread and activates the MAC layer within the base thread to fetch incoming bytes from the queue. The MAC fetches bytes from the queue and assembles them into a packet. If the destination address is a broadcast address or matches the local node's address, the MAC layer uses the destination port in the packet to find the local thread that is blocking waiting for this packet.

The network layer can either be implemented as part of the base network thread or can be implemented separately in another thread. In the latter case, the base thread will post the semaphore on which network layer thread is blocking, and then the packet will be copied from thread to thread. In general, if the upper layer is implemented outside of the lower layer's thread, then a packet will be copied between two user-level threads.

The advantage of a multi-threaded user-level network stack is that it promotes flexibility, at a cost in performance. In comparison, systems with a monolithic network stack implemented as part of the kernel are relatively inflexible. Suppose an application designer did not know in advance which features out of multicast, broadcast, or unicast are needed in a sensor network deployment. In systems with a monolithic network stack, the designer in the worst case is forced to load all three network layer modules at run time, leading to inefficient usage of highly limited RAM (less than 4 KB on ATMEL microprocessors [7]).

The MANTIS decision to implement the network stack as one or more user-level threads allows a designer to activate or deactivate a particular routing protocol or reliable protocol on demand as a user-level thread. For example, if a routing protocol has been stored in flash, then that protocol can be activated by simply starting it in RAM. If needed, multiple routing protocol threads can coexist at the same time. Each packet is directed to the appropriate protocol thread on a per-packet basis. This flexible structure is especially useful for dynamic reprogramming in sensor networks, enabling application developers to dynamically reprogram network functionality such as routing in deployed sensor nodes by starting, stopping, and deleting user-level threads.

The decision to implement a network stack as a set of user-level threads is also useful for cross-platform prototyping of network stack functionality on X86 PCs prior to deployment in WSNs, as described in the next section.

The MANTIS system enables zero copies within a thread and single copies across threads. If a network stack is wholly implemented within the base network stack thread, then our approach begins to resemble the zero copy approach of TinyOS, SMAC [10] and zero copy sockets [11]. At present, we are investigating zero copies across multiple threads.

In MANTIS, the header and the data buffer for a packet are allocated separately. The MANTIS network stack allows each layer to define its own header structure. Each header consists of

two parts, i.e. a common header and a protocol header. The structure of the common header is static while the structure and length of the protocol header could be varied.

The MANTIS network stack occupies about 200 bytes of RAM, when only the MAC and physical layers of the base network thread are considered. Two system data queues are required of 64 bytes each. The rest of the space is largely consumed by low-level configuration parameters for the CC1000 radio. Modules for a broadcast flooding routing protocol and a simple stop and wait protocol are provided in MOS as default examples for developing protocols at the network and application layers. Network layer broadcast flooding adds an additional thirty bytes of RAM.

The following APIs are provided for connectionless packet I/O:

```
mos_send(char* header, char headerLen, char* data, char dataLen);
```

```
mos_send_to(uint16_t addr, uint8_t port, char* data, char dataLen, uint8_t proto);
```

```
mos_recv(Packet* pkt, uint8_t port, uint8_t proto);
```

The network stack also allows the application to specify the length of time that the thread is willing to be blocked on the mos\_recv() function in order to avoid waiting indefinitely.

Together, the code size of the kernel, scheduler, and network stack occupies less than 500 bytes of RAM and about 14 KB of flash. This permits sufficient space for multiple application threads to execute in the ATMEL's 4 KB of RAM, as well as sufficient storage in the ATMEL's 128 KB of flash storage.

## 2.5 Device Drivers

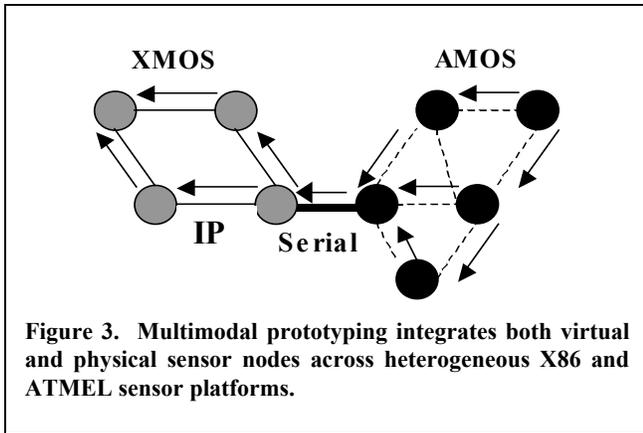
MANTIS adopts the traditional logical/physical partitioning with respect to device driver design for the hardware. For example, to turn the green LED on, the LED system API provides a led\_green\_on call (logical level), which is transformed to a PORTA |= 0x80 (physical level) action. The application developer need not interact with the hardware to accomplish a given task. However, full access is available to the hardware for the adventurous.

## 3. ADVANCED SENSOR-SPECIFIC FEATURES OF MANTIS OS

Sensor networks impose additional unique demands on the design of operating systems beyond lightweight resource constraints. Sensor networking application developers need to be able to prototype and test applications prior to distribution and physical deployment in the field. Also, during deployment, in-situ sensor nodes need to be capable of being both dynamically reprogrammed and remotely debugged. In the next sections, MANTIS identifies and implements each of these three key advanced features for expert users of general-purpose sensor systems.

### 3.1 Multimodal Prototyping Environment

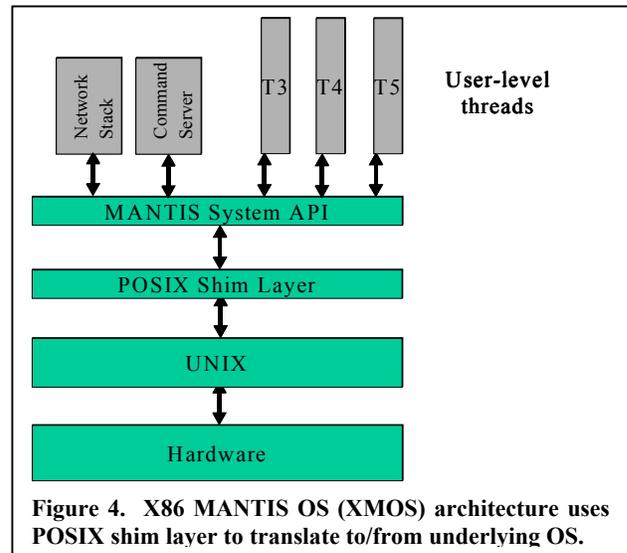
The MANTIS prototyping environment provides a framework for prototyping diverse applications across heterogeneous platforms. A key requirement of sensor systems is the need to provide a



prototyping environment to test sensor networking applications prior to deployment. Postponing testing of an application until after its deployment across a distributed sensor network can incur severe consequences. As a result, a prototyping environment is an especially helpful tool for sensor network application developers.

The MANTIS prototyping environment extends beyond simulation to provide a larger framework for development of network management and visualization applications as virtual nodes within a MANTIS sensor network. First, MANTIS has the desirable property of enabling an application developer to test execution of the same C code on both virtual sensor nodes and later on in-situ physical sensor nodes. Second, MANTIS seamlessly integrates the virtual environment with the real deployment network, such that both virtual and physical nodes can coexist and communicate with each other in the prototyping environment, as shown in Figure 3. Seamless integration enables phased deployment and testing of an application, i.e. application code could first be evaluated on an all-virtual network, then be deployed without modification to a hybrid network of both virtual and a few physical nodes, followed by full deployment on an all-physical network. The combination of all-virtual, hybrid, and all-physical nodes of testing form a *multimodal* prototyping environment. Third, MANTIS permits a virtual node to leverage other APIs outside of the MANTIS API, e.g. a virtual node with the MANTIS API could be realized as a UNIX X windows application that communicates with other rendering or database APIs to build visualization and network management applications, respectively. This virtual node, a.k.a. UNIX application, would incorporate the MANTIS system API as a simple means of becoming just another node within the MANTIS network of virtual and physical nodes.

MANTIS achieves a multimodal prototyping environment by preserving a common C API across all platforms. This approach resembles WINE [12], but eliminates the problems of hidden system calls, since all such calls are publicly known in MANTIS. Due to the wide availability and support for the AVR microcontroller under Linux and Windows, it is possible to build MOS, with minor modifications, as an application that runs on the X86 platform over both Linux and Windows. We call this user space application running on an X86 platform XMOS. For example, Figure.4 illustrates XMOS utilizing a POSIX shim layer to translate between MANTIS' uniform API and the underlying



UNIX operating system. In this way, MOS applications can be realized as both virtual sensor nodes on X86 platforms as well as live applications on ATMEL sensor nodes (AMOS). This enables MANTIS to support multimodal networks, consisting of XMOS nodes and AMOS nodes seamlessly interacting with each other. The same C source code runs transparently over both XMOS and AMOS platforms, enabling phased deployment from XMOS to AMOS. Figure 3 shows the structure of the network, with the two networks connected to each other via a serial RS232 link. Thus, a `mos_send(...)` system call on the AMOS nodes causes the data to be transmitted over the radio. The bridge nodes on either side of the bridging serial link would additionally send the data over the serial link using the `mos_uart_send(...)` call. A `mos_send(...)` call on the XMOS nodes causes the data to be transmitted over the IP network instead.

The structural implications of the above multimodal prototyping environment afford great flexibility to application developers. First, XMOS nodes need not be identical and indeed heterogeneous applications can be supported simultaneously. For example, some XMOS nodes can be written as base stations, while others may perform aggregation duties for directed diffusion [13], and still others may coexist to perform multicast routing [14]. Second, XMOS nodes are not confined to a single PC, and can be distributed across any number of PCs, maintaining communication via IP packets. This eases the ability of the prototyping environment to scale to large numbers of XMOS virtual nodes. Third, an arbitrary number of bridging links can connect XMOS and AMOS environments, and need not be limited to serial links either. Fourth, virtual nodes must support but are not limited to the MANTIS API. As a result, a virtual node realized as a UNIX application could be integrated into the MANTIS sensor network on one side and speak with a rendering API, database API, X windows API, or socket API on another side. Thus, the sensor network can be accessed from any virtual node, easing development of applications for visualization, network management, and gateway translation to other networks. The gateway function is especially critical to translate sensor packet data to/from IP networks. Fifth, since the network stack is implemented as user-level thread(s) above the common API, then

an added bonus is that the XMOS environment can be used to prototype OS functionality in the form of networking routing and reliability functions. XMOS is not confined to prototyping user programs only. Finally, provided that hardware translation is correct, the XMOS architecture offers the potential to feed real sensor data into virtual nodes to drive prototype evaluation.

A variety of other sensor networking simulators possess some but not all of the features of the MANTIS multimodal prototyping environment. TOSSIM is a simulator for TinyOS [15], and enables the same code to run in PC simulation as on real sensor nodes, enabling debugging and verification on PCs prior to deployment. However, the simulator has to run on one machine and with the same application instance inside. TOSSF extends TOSSIM to enable heterogeneous applications, but they're still confined to one PC [16]. Sensorsim is an extension to ns2 and provides a simulation framework which models the sensor nodes and also provides a "hybrid" simulation combining the real and virtual network [17]. However, the sensor network applications are required to be re-implemented for the target platform, resulting in two completely different code bases that must be maintained. emStar is a framework for developing applications on wireless sensor networks and combines pure simulation, hybrid mode and real distributed deployments [3]. However, the implementation is based on the combination of HP iPAQ platform and the motes. In MANTIS, no extra hardware such as the iPAQ is required.

The MANTIS multimodal framework does have some limitations. By choosing to preserve a high-level API across platforms rather than low-level instructions as in a virtual machine, each XMOS node does not perfectly model the performance of a sensor node. Our tradeoff has been for improved flexibility rather than precise emulation. Also, not all OS functionality can be tested in the above architecture. While the network stack and remote shell via the command server can be tested, as well as user programs, other functionality such as the kernel's scheduler are at present beyond the cross-platform testing capabilities of XMOS.

### **3.2 Dynamic Reprogramming**

Dynamic reprogramming or retasking is an especially useful feature for sensor networks. Research has found that sensor nodes should be remotely reconfigurable over a wireless multi-hop network after being deployed in the field [18]. Since sensor networks may be deployed in inaccessible areas and may scale to thousands of nodes [19], this simplifies management of the sensor network, i.e. so that biologists need not go into the field again to reprogram sensors and change parameters such as the sensor's sampling rate and trigger threshold or algorithms such as sensor calibration or time synchronization.

MOS achieves dynamic reprogramming on several granularities: reflashing of the entire OS; reprogramming of a single thread; and changing of variables within a thread. Another feature that is especially useful for sensor systems is the ability to remotely debug a running thread. MOS provides a remote shell that enables a user to login and inspect the sensor node's memory, e.g. the thread table of an executing thread.

To overcome the difficulty of reprogramming the network, MOS includes two reprogramming modes. The simpler programming mode is similar to that used in many other systems and involves direct communication with a specific MANTIS node. On a

Nymph, this would be accomplished via the serial port: The user simply connects the node to a PC and opens the MANTIS shell. Upon reset, MOS enters a boot loader that checks for communication from the shell. At this point, the node will accept a new code image, which is downloaded from the PC over the direct communication line. From the shell, the user also has the ability to inspect and modify the node's memory directly (peek and poke), as well as spawn threads and retrieve debugging information—including thread status, stack fill, and other such statistics—from the operating system. The boot loader transfers control to the MOS kernel on command from the shell, or at startup if the shell is not present.

The more advanced programming mode is used when a node is already deployed, and does not require direct access to the node. The spectrum of dynamic reprogramming of in-situ sensor networks ranges from fine grained reprogramming (modifying constants like sampling rate) to complete reprogramming of the sensor nodes. MOS has a provision for reprogramming any portion of the node—up to and including the operating system itself—while the node is deployed in the field. This is accomplished through the MOS dynamic reprogramming interface. The capability to use the dynamic reprogramming interface will be built into the MANTIS programming tool.

Current solutions for dynamic reprogramming [20] are virtual machine (VM) -based where the VM resides over the underlying sensor operating system and processes the incoming code capsules. A special stack-based instruction set is used to reprogram the sensor nodes, reducing the amount of data that is transmitted over the network. In contrast to the VM based approach, MOS allows binary updates to reprogram a node. The developer does not need to learn a new stack-based instruction set; instead, the existing deployed application only needs to be modified and recompiled, then a binary patch may be transmitted to the MANTIS node.

The dynamic reprogramming capability is actually implemented as a system call library, which is built into the MOS kernel. Any application may write a new code image through calls to this library; the code image is stored into EEPROM as it is written. The application then calls a commit function that writes out a control block for the MOS boot loader, which causes it to install the new code on reset. A software reset completes the reprogramming process. Using the reprogramming library, an application—such as the MANTIS command server—may download a patch using any communications method it desires (typically the regular network stack), apply the patch to the existing code image, and run the updated code. Thus, the entire code image, with the exception of the locked boot loader section, may be reprogrammed over an arbitrary network while the node is deployed.

### **3.3 Remote Shell and Command Server**

Traditional solutions for network management such as SNMP [21] are not applicable to highly dynamic sensor networks. Existing solutions for monitoring sensor networks look at topology extraction [22] and computing summaries of network properties for energy efficient monitoring of sensor networks [23]. In addition to these mechanisms, the user may wish to manage the nodes in the network in other ways. To provide this flexibility, MOS includes the MANTIS Command Server (MCS).

From any device in the network equipped with a terminal (a laptop PC, for example), the user may invoke the command server client (also referred to as the shell) and “log in” to a node. This node may be either a physical node (e.g. on a Nymph or Mica board) or it may be a virtual node running as a process on a PC. Figure 5 illustrates an example of the remote shell interface.

The MCS itself is implemented as an application thread. It listens on a network port for commands and replies with the results, in a manner similar to RPC. In effect, the shell gains the ability to control a node remotely through MCS. The user may alter the node's configuration settings, run or kill programs, display the thread table and other operating system data, inspect and modify the node's data memory, and call arbitrary user-defined functions. The shell is a powerful debugging tool, since it allows the user to

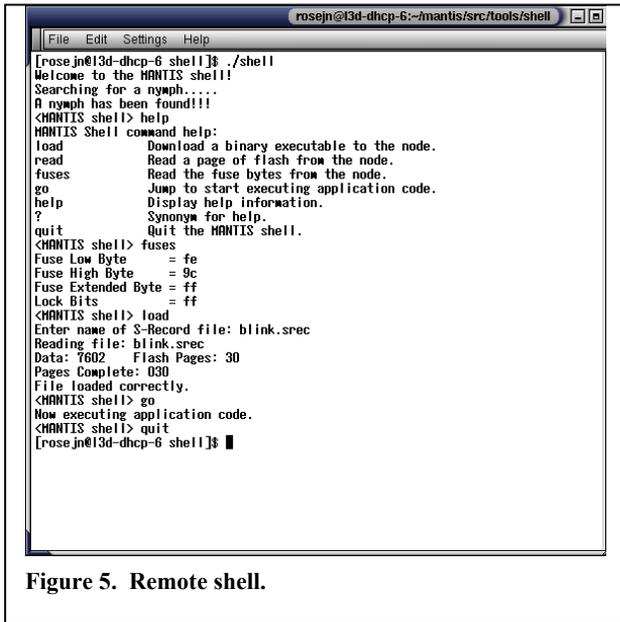


Figure 5. Remote shell.

examine and modify the state of any node, without requiring physical access to the node.

#### 4. MANTIS HARDWARE

The MANTIS hardware nymph's design was inspired by the Berkeley MICA and MICA2 Mote architecture [1]. To help lower our development costs, shorten our development cycle, and enhance our research goals, we designed the MANTIS hardware nymph sensor node, adhering to the same themes of ease of use, flexibility, and adaptation to sensor networks that characterized our software design. The learning curve for novice users is lowered by employing a single-board design, as shown in Figure 6, altogether incorporating a low power Atmel Atmega128(L) microcontroller (MCU) [7], analog sensor and digital ports, a low power Chipcon CC1000 multi-channel RF radio [24], EEPROM, power ADC sensor, and serial ports on a quad-layer 3.67 x 3.3 cm Printed Circuit Board (PCB). For the common user, the single-board design eliminates the need for a separate sensor board or separate programming board, which reduces volume and cost. The pins for the serial interface are directly accessible on the nymph in a standard DIP package, enabling direct connection of each nymph to a laptop via a serial cable, as shown in Figure 6.

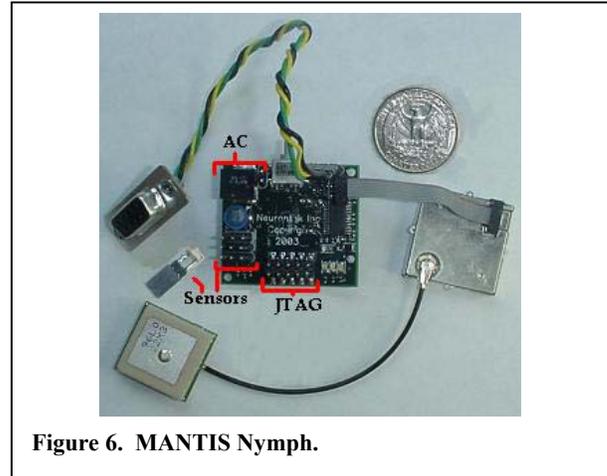


Figure 6. MANTIS Nymph.

Direct serial accessibility combined with dynamic reprogramming over wireless largely eliminate the need for a programming board for the common user. Nymphs are versatile in that any node can serve as a base station or as a leaf. In addition, three sensor interfaces are built into each nymph and are directly accessible to the user via wire-wrappable DIP pins, eliminating the need for the sensor board in the common case. A standard three-wire interface similar to the popular Lego Mindstorms was selected, enabling a novice to quickly prototype from a large selection of inexpensive resistive sensors. Also, GPS capability has been added to each nymph in the form of a connector that fits the Trimble Lassen SQ GPS chip shown to the right of the nymph in Figure 6. Again, the goal is to simplify deployment of GPS-enabled applications for beginning users. If the GPS chip is not needed, then the connector is simply vacant. Finally, the nymph includes an AC/DC option. This is useful for prototyping in the lab and avoids excessive consumption of batteries. An AC/DC adapter from Radioshack is satisfactory. A simple 3-way switch toggles between the AC/DC option, OFF and the battery option. We envision that the power option will be useful in future deployments of indoor sensor networks, where power outlets are readily available for exploitation.

To support advanced research, the nymph includes several interfaces that allow expert users to extend its capability. First, the nymph exports a standard sized JTAG DIP interface for expert users that need to burn the bootloader into the Atmel's flash. For example, researchers experimenting with dynamic reprogramming may need to reset the fuses on the flash. For the novice user, we envision that the bootloader will be preinstalled by the manufacturer or an expert user with access to a JTAG programming device. In difficult debugging situations, the JTAG interface can also be used for line by line, in-system debugging using GDB. Second, the nymph includes a 20-pin connector with standard DIP interface for wire-wrapping or development of an advanced add-on boards with mating connector. This connector has direct access to the MCU's external interrupt pins, I<sup>2</sup>C bus, data lines, timers, and pulse width modulation (PWM) pins. Some potential add-on boards would be I<sup>2</sup>C expanders that use the interrupt and I<sup>2</sup>C pins to add touch pads for example. The data lines may be used to add liquid crystal displays, while the PWM pins may be used for controlling motors, timers for time sensitive applications, or simply as more pins for general digital I/O.

Third, the MANTIS nymph supports multiple antenna options, including the addition of an antenna amplifier, via another connector. This connector acts more like a jumper enabling and disabling the built in low-range low power capabilities and replacing them by add-on circuitry. The add-on circuitry implements a 30dB low-noise power amplifier that is a 24-pin chip plus its additional support circuitry and properly matched 915 MHz antenna. The addition of the amplifier increases the communication range of the MANTIS Nymph to up to 2km at the cost of up to half a Watt additional power consumption. For those reasons we provide the connector as an option and not a requirement. One final important advanced feature is the addition of a single channel I<sup>2</sup>C 16-bit ADC. This ADC enables monitoring of the battery voltage level.

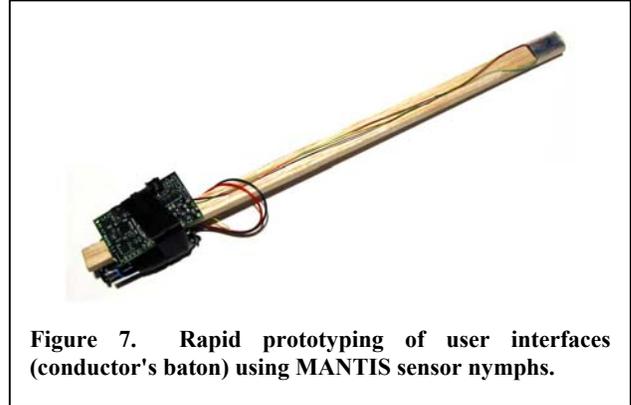


Figure 7. Rapid prototyping of user interfaces (conductor's baton) using MANTIS sensor nymphs.

Test Reformed	Input Voltage (V)	Load Current (mA)
Just Operating System Running	30	13
Scheduled Single Thread While()	30	14
Scheduler with blinking LED ON	30	22
Scheduler with blinking LED OFF	30	14
Reading Sensor Data on while()	30	14
Sensing and Sending over Serial at 192k baud	30	14
Sensing and Sending over radio, Transmitting at max power	30	73
Sensing and Sending over radio, Transmitting at min power	30	40
Sensing and Sending over radio, Receiving at max power	30	40
Sensing and Sending over radio, Receiving at min power	30	27
Absolute Max power with all LEDs on and radio transmit at max power	30	98
Absolute Max power with all LEDs on and radio receive at max power	30	60
Single LED power consumption	30	9
Everything in sleep mode	30	under 1
Elite Nymph - Operating System and GPS	30	98
Elite Nymph - Everything Running at maximum power including LEDs	30	172

Table 1. Power consumption of MANTIS Nymph in various modes of operation.

Power consumption numbers for the 4-layer nymph are given in Table 1. GPS consumes significant power and will require careful power management to limit its impact on battery lifetime. Comparable recent hardware technology includes the MICA2 Notes [25] and the GPS-enabled GNOMES [26].

## 5. MANTIS APPLICATIONS

### 5.1 Rapid Prototyping of User Interfaces

The MANTIS platform has successfully served as the basis for a system for constructing rapid prototypes of wireless input devices and user interfaces [6]. The conductor's baton, shown in Figure 7, is an example of a gestural interface device that utilizes a two-axis accelerometer to detect the motion of the user's hand. Using one nymph to read the sensor values from the accelerometer and a second nymph (connected to a PC via the serial port) as a base station to wirelessly collect the data, a musical tempo is interpreted from the beats marked by the user. This tempo is then transmitted to an algorithmic music application, which changes the speed at which the music is played based on the user's gestures.

### 5.2 GPS-Enabled Time Synchronization

To demonstrate the capabilities of MOS and the MANTIS nymph, we implemented and evaluated a variety of time synchronization algorithms in a GPS-enabled framework. GPS-enabled nymphs provide a framework for evaluating the in-situ accuracy of deployed time synchronization algorithm. Prior to the advent of GPS-enabled nymphs, the accuracy of time synchronization was evaluated in more cumbersome ways in sensor networks, either via equipment probes, or outside of the sensor network realm altogether. GPS has also been used in sensor networks for location awareness [26] and location tracking of wildlife, e.g. Zebrant [27].

Time synchronization is important for sensor networks in order to correctly timestamp an event and coordinate wake/sleep duty cycles of low power sensor networks [28,29,30,31]. Figure 8 illustrates a general network topology in which nodes that either lack GPS or lack line of sight to be able to observe GPS satellites request clock synchronization with GPS-enabled nodes. We constructed an experiment consisting of a network topology with five sensor nymphs in a row. This linked routing chain is similar to the linear route in Figure 8 taken by node N2's Simple Network Time Protocol (SNTP) request [32], which propagates to the GPS-enabled node four hops away. On a given sensor node, the estimated clock obtained from a time synchronization algorithm is compared to the authoritative GPS clock right after reception of query response. Sync-requests are initiated by the client every 10 seconds. This experiment was repeated a hundred times to obtain

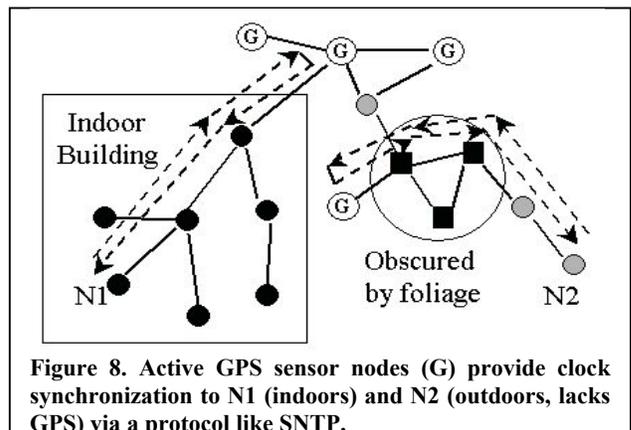


Figure 8. Active GPS sensor nodes (G) provide clock synchronization to N1 (indoors) and N2 (outdoors, lacks GPS) via a protocol like SNTP.

an average estimate of the inaccuracy. SNTP provides a simple and well-known mechanism for a client to synchronize with a server who has an accurate clock by using the roundtrip delay over multiple hops. An adaptation of SNTP [31] that employs hierarchy was also implemented, i.e. each child uses SNTP on its parent. We call this approach hierarchical SNTP (HSNTP). Both algorithms are also evaluated over multi-frequency radios, creating MSNTP and MHSNTP. As MANTIS supports communicating in multi-channels, by employing a separate control channel for time synchronization queries and responses, variations in latency due to collisions are reduced, resulting in more accurate clock estimation. Table 2 verifies that multi-channel radios dramatically improve time synchronization accuracy and lower deviation. Our anecdotal experience with GPS was that on power up GPS takes from 3 to 10 minutes to compute its position, and less than 30 seconds from a warm start. GPS was accurate to 10 meters 90% of the time.

Algorithm	Mean Error	StdDev
SNTP	39.81979	43.2736
HSNTP	23.6972	26.438
MSNTP	20.9021	22.9463
MHSNTP	10.44147	11.5746

Table 2. Time synchronization accuracy in microseconds.

## 6. FUTURE WORK

The MANTIS system is still very much a work in progress. While we have managed to achieve a lightweight OS, there are additional demands on sensor OS designs that have not yet been fully integrated into our design. We have identified low power operation as a key future direction for development of the MANTIS OS. The challenge will be to integrate within a multi-threaded model the idea of sleeping threads until useful work is required. First, the idle loop must be replaced with a power-efficient scheduler. One approach is to add another parameter to the `thread_new` call so that the application can provide static hints at the time it is spawned. A follow-on approach would incorporate dynamic hints from within the application with a `power_hint` call to modify the application's requirements dynamically. Prior work on power-efficient scheduling and systems should be leveraged [33,34]. Additional complications will result from integrating components such as the Atmel and CC1000 with multiple low power modes. At present, MOS exports setting these modes through the API, but applications have not yet been developed to exploit these low power features. We are further interested in pushing the power-efficient scheduler into user space to further streamline the kernel, similar to the micro-kernel architecture [35].

There is still some work to be done in demonstrating reliability for code updates over the network, optimizing the size of updates, and ensuring the security and authenticity of updates. Even after those issues are addressed, we have only solved the problem of reprogramming a single node remotely. While one could certainly iterate through all nodes in a network in order to reprogram them all, that would be inefficient and perhaps infeasible if the network were large. The broader problem of remotely reprogramming a *network*, as opposed to a node, will be addressed in future work.

We also intend to integrate security into dynamic reprogramming, so that downloaded code can be authenticated, decrypted, and checked for tampering. At present, we have implemented an RC5-based CBC mode block cipher encryption/decryption library. This library also provides functions for sending encrypted packets and generating message authentication codes to protect integrity of packets. The API is:

```
mos_sec_send_to(uint16_t addr, uint8_t port, char* data, char
dataLen, uint8_t proto, rc5key_info *rc5key);

mos_sec_rcv(Packet* pkt, uint8_t port, uint8_t proto,
rc5key_info *rc5key);
```

The overhead of this security library is very small, about 110 bytes of RAM. The encrypted packet transmission function adds about 6% delay compared to non-encrypted packet transmission.

An area that has not yet been addressed is simulating the wireless channel within the multimodal prototyping environment. One challenge is the difficulty of simulating wireless communication channels, especially indoor communication. Another challenge is building a structure that enable media contention among multiple virtual nodes.

The MANTIS project was recently awarded an NSF SENSORS grant to study the role of sensor networks in fighting forest fires Stay tuned to the MANTIS Web site <http://mantis.cs.colorado.edu>

## 7. CONCLUSION

The MANTIS sensor system achieves a lightweight classically structured multithreaded operating system in a memory footprint of less than five hundred bytes, including kernel and network stack. The MANTIS OS supports a simple C API that enables a shallow learning curve, cross-platform support, and reuse of a large installed code base. MOS also supports advanced sensor-specific features such as multimodal prototyping, dynamic reprogramming, and remote shells. The MANTIS nymph offers a single-board GPS-enabled solution that is also extensible.

## 8. REFERENCES

- [1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System Architecture Directions For Network Sensors", *ASPLOS* 2000, pp. 93-104.
- [2] R. Min, M. Bhardwaj, S. Cho et al, "An Architecture for a Power-Aware Distributed Microsensor Node," in *IEEE Workshop on Signal Proc. Systems*, pp. 581-590, Oct 2000.
- [3] J. Elson, S. Bien, N. Busek, V. Bychkovskiy, A. Cerpa, D. Ganesan, L. Girod, B. Greenstein, T. Schoellhammer, T. Stathopoulos, and D. Estrin: "EmStar: An Environment for Developing Wireless Embedded Systems Software" *CENS Technical Report 0009*, March 24, 2003.
- [4] F. Martin, B. Mikhak, and B. Silverman, "MetaCricket: A designer's kit for making computational devices," *IBM Systems Journal*, vol. 39, nos. 3 & 4, 2000.
- [5] R. L. Rivest. "The RC5 Encryption Algorithm", *Proceedings of the 1994 Leuven Workshop on Fast Software Encryption*, pages 86-96
- [6] J. Carlson, R. Han, S. Lao, C. Narayan, S. Sanghani, "Rapid Prototyping Of Mobile Input Devices Using Wireless Sensor

- Nodes", 5th IEEE Workshop On Mobile Computing Systems and Applications (WMCSA) 2003 (to appear).
- [7] Atmel AVR 8-bit RISC processor, <http://www.atmel.com/products/AVR>
- [8] Portable Operating System Interface (POSIX) - Part 1: System Application Programming Interface (API) [C Language] ISO/IEC 9945-1:1996, IEEE
- [9] D. Ely, S. Savage, and D. Wetherall: "Alpine: A user-level infrastructure for network protocol development". In *Proc. 3rd USENIX Symposium on Internet Technologies and Systems*, pages 171-183, March 2001.
- [10] Wei Ye, John Heidemann and Deborah Estrin, "An Energy-Efficient MAC Protocol for Wireless Sensor Networks", In *Proceedings INFOCOM*, New York, NY, USA, June, 2002.
- [11] H. K. Jerry Chu, "Zero-Copy TCP in Solaris", *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, January 1996.
- [12] WINE, <http://www.winehq.com/>.
- [13] C. Intanagonwiwat, R. Govindan, D. Estrin, "Directed Diffusion," *MobiCom 2000*, pp. 56-67
- [14] A. Sheth, B. Shucker, R. Han, "VLM2: A Very Lightweight Mobile Multicast System for Wireless Sensor Networks", *IEEE Wireless Communications and Networking Conference (WCNC) 2003*, New Orleans, Louisiana.
- [15] P. Levis and N. Lee. Simulating tinyos networks. <http://www.cs.berkeley.edu/pal/research/tossim.html>.
- [16] L. F. Perrone and D. M. Nicol: A Scalable Simulator for TinyOS Applications, Winter Simulation Conference, 2002.
- [17] S. Park, A. Savvides, M. B. Srivastava, "SensorSim: A Simulation Framework for Sensor Networks", In *the Proceedings of MSWiM 2000*, Boston, MA, August 11, 2000.
- [18] A. Mainwaring, J. Polastre, R. Szewczyk D. Culler, J. Anderson, "Wireless Sensor Networks for Habitat Monitoring", *First ACM Workshop on Wireless Sensor Networks and Applications (WSNA) 2002*, pp. 88-97.
- [19] S. Tilak, N.B. Abu-Ghazaleh, W. Heinzelman, "A taxonomy of wireless micro-sensor network models", *ACM SIGMOBILE Mobile Computing and Communications Review*, Vol. 6 Ch. 2 pages 28-36. 2002.
- [20] P. Levis, D. Culler "Mate: a Virtual Machine for Tiny Networked Sensors" *ASPLOS*, Oct. 2002.
- [21] J. D. Case, M. Fedor, M. L. Schostall, and C. Davin. *RFC 1157: Simple network management protocol (SNMP)*. RFC, IETF, May 1990
- [22] B. Deb, S. Bhatnagar, B. Nath "A Topology Discovery Algorithm for Sensor Networks with Applications to Network Management", *DCS Technical Report DCS-TR-441, Rutgers University* May 2001
- [23] J. Zhao, R. Govindan, D. Estrin "Computing Aggregates for Monitoring Wireless Sensor Networks", *First IEEE International Workshop on Sensor Network Protocols and Applications*, Anchorage, AK. May 2003
- [24] Single chip ultra low power RF transceiver [http://www.chipcon.com/files/CC1000\\_Data\\_Sheet\\_2\\_1.pdf](http://www.chipcon.com/files/CC1000_Data_Sheet_2_1.pdf), 2001
- [25] Crossbow, <http://www.xbow.com/>.
- [26] E. Welsh, W. Fish, P. Frantz, "GNOMES: A Testbed for Low-Power Heterogeneous Wireless Sensor Networks," *IEEE International Symposium on Circuits and Systems (ISCAS)*, Bangkok, Thailand, 2003.
- [27] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. "Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebnet", In *ASPLOS*, San Jose, CA, October 2002.
- [28] J. Elson, D. Estrin "Time Synchronization for Wireless Sensor Networks" *International Parallel and Distributed Processing Symposium (IPDPS), Workshop on... Wireless and Mobile Computing*, April 2001
- [29] J. Elson, L. Girod, D. Estrin "Fine-Grained Network Time Synchronization using Reference Broadcasts", In *OSDI 2002*, Boston, MA. December 2002.
- [30] J. Elson, K. Römer, "Wireless Sensor Networks: A New Regime for Time Synchronization", in proceedings of the *First Workshop on Hot Topics In Networks (HotNets-I)*, Princeton, New Jersey. October 28-29 2002
- [31] S. Ganeriwal, R. Kumar, S. Adlakha, M. Srivastava, "Network-wide Time Synchronization in Sensor Networks," *Technical report, UCLA, Dept of Electrical Engineering*, 2002.
- [32] Simple Network Time Protocol, (SNTP) version 4. IETF RFC 2030
- [33] D. Grunwald, C. B. Morrey III, P. Levis, M. Neufeld, K. Farkas, "Policies for Dynamic Clock Scheduling", *Operating Systems Design and Implementation - 2000*.
- [34] W. Hamburger, D. Wallach, M. Viredaz, L. Brakmo, C. Waldspurger, J. Bartlett, T. Mann, K. Farkas, "Itsy: Stretching the Bounds of Mobile Computing," *IEEE Computer*, vol. 34, no. 4, April 2001, pp. 28-36.
- [35] D.R. Engler, M. Frans Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-level Resource Management", *Symposium on Operating Systems Principles (SOSP)*, December 1995, pp. 251-266.