MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms

Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, Richard Han
University of Colorado at Boulder
Computer Science Department
Boulder, CO, 80309
rhan@cs.colorado.edu

ABSTRACT

The *MANTIS MultimodAl system for NeTworks of In-situ wireless Sensors provides a new multithreaded cross-platform embedded operating system for wireless sensor networks. As sensor networks accommodate increasingly complex tasks such as compression, aggregation and signal processing, preemptive multithreading in the MANTIS sensor OS (MOS) enables micro sensor nodes to natively interleave complex tasks with time-sensitive tasks, thereby mitigating the bounded buffer producer-consumer problem. To achieve memory efficiency, MOS is implemented in a lightweight RAM footprint that fits in less than 500 bytes of memory, including kernel, scheduler, and network stack. To achieve energy efficiency, the MOS power-efficient scheduler sleeps the microcontroller after all active threads have called the MOS sleep() function, reducing current consumption to the μA range. A key MOS design feature is flexibility in the form of cross-platform support and testing across PCs, PDAs, and different micro sensor platforms. Another key MOS design feature is support for remote management of in-situ sensors via dynamic reprogramming and remote login.*

Keywords: embedded operating system, sensor networks, multithreaded, lightweight, low power, cross-platform, dynamic reprogramming

1. INTRODUCTION

The popularity of wireless sensor networks (WSNs) as an important new research domain has grown dramatically [1] [2]. WSN systems typically consist of resource-constrained micro sensor nodes that self-organize into a multi-hop wireless network. This sensor network monitors the environment, collects sensed data and relays the data back to a collection point typically residing on the Internet. WSNs integrate hardware platforms, embedded operating systems, networked communication, and backend data services together into a complete system capable of providing novel distributed in-situ sensing of environmental phenomena. Standard micro sensor systems include Berkeley's Mote/TinyOS architecture [3], MetaCricket [30], MIT's location-aware cricket [31], CU-Boulder's MANTIS system [5], Europe's Smart-Its [33], Eyes [34], and BTNodes [32] projects.

This paper investigates the practicality of implementing the popular approach of preemptively time-sliced multithreading on today's micro sensor nodes, and explores how such a system could be adapted to the characteristics of WSNs. A thread-driven approach is attractive in sensor networks for many of the same reasons that it has been adopted for PDAs, laptops, and servers. In a thread-driven system, an application programmer need not be concerned about indefinitely blocking or being indefinitely blocked by other tasks during execution, except for shared resources, because the scheduler will preemptively time-slice between threads, allowing some tasks to continue execution even though others may be blocked. This automated time-slicing considerably simplifies programming for an application developer. This paper demonstrates that the added OS complexity needed to support preemptive time-slicing is easily accommodated in today's MICA2 motes, with a kernel memory cost of the less than 500 bytes, including the scheduler and network stack. Moreover, the paper shows that multithreading and energy efficiency are not mutually exclusive, i.e. that a multithreaded system can be designed to sleep efficiently when application threads indicate that there is no useful work to be done.

The finely interleaved concurrency of multithreading is useful in sensor systems to prevent one long-lived task from blocking execution of a second time-sensitive task. Sensor networks are being tasked to perfom increasingly complex duties such as signal processing and collaborative target tracking, time synchronization [43] [45], localization [31], compression/aggregation [37], and encryption. As we will show later in Section 3, such tasks can be long-lived enough in a single-threaded run-to-completion system to prevent time-sensitive processing of other tasks, e.g. processing of radio packets by different layers of the network stack. If the network stack is blocked from fully processing arriving packets until the long-lived task runs to completion, then the network stack's bounded buffers could quickly overflow, especially in sensor nodes with limited RAM, resulting in lost packets. Multithreading conveniently mitigates this classic bounded buffer producer-consumer problem by interleaving processing of packets by the network stack thread with execution of multiple long-lived complex tasks, so that packets are emptied from the buffer before overflow is reached. Our discussion focuses on a loose interpretation of the bounded buffer problem in terms of its resource constraints rather than its more traditional interpretation in the field of synchronization.

The challenges of designing a multithreaded embedded operating system for WSNs are motivated by the severe resource constraints imposed by micro sensor nodes, e.g. their limited run-

time memory as well as their limited energy lifetimes. The runtime RAM available on micro sensor nodes is exceedingly scarce, e.g. 4 KB for today's MICA2 motes [9]. While sensor nodes may diversify towards nano nodes and macro nodes with lesser and greater capabilities, this current generation of micro sensor nodes is the standard starting reference that is assumed in this paper. Because of these severe memory constraints, traditional multithreaded embedded operating systems such as QNX and VXWorks occupy too large of a memory footprint to execute on micro sensor nodes [3], with embedded Linux facing the same limitations. Two embedded real-time embedded operating systems, AVRX [13] and μCOS [14], have been written for the AVR microcontroller found on the MICA2 motes. Both achieve preemptive multitasking in a lightweight RAM footprint of less than 4 KB. μCOS is a licensed commercial OS, while AVRX is open source. The MANTIS open source OS (MOS) differs from these two embedded RTOSs by being adapted to the additional requirements imposed by sensor networks, e.g. the development of a power-efficient scheduler to reduce energy consumption and the implementation of advanced sensor-specific features like remote dynamic reprogramming of micro sensor nodes.

In addition to memory efficiency, micro sensor nodes also require energy efficiency in the design of the sensor OS. Micro sensor nodes are often deployed in-situ apart from the electrical power grid, and therefore rely on battery power or energy harvesting, e.g. solar cells. Given a set of new AA batteries, the lifetime of such nodes can be extended to a few months depending upon the extent to which the duty cycle is lowered [26]. Key new challenges in the design of a thread-driven sensor OS therefore include achieving both a lightweight memory footprint as well as energy-efficient operation.

This paper describes MANTIS OS, a lightweight and energy-efficient multithreaded operating system for MultimodAl NeTworks of In-situ micro Sensor nodes. At present, the MOS kernel is able to achieve multithreaded preemptively 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. In addition, MANTIS OS is designed to provide cross-platform support across PC's, PDAs, as well as diverse micro sensor hardware platforms. For example, MANTIS OS currently supports both the MICA2 motes as well as the MANTIS nymph. MANTIS OS also seeks to provide tools to ease deployment and management of in-situ sensor networks.

In order to achieve cross-platform support, MOS was designed to leverage the properties of a portable standard programming language, in this case the C programming language. MOS enables the same application code to execute on a variety of platforms, ranging from PC's to PDA's to different micro sensor platforms. As detailed in our earlier work [5], this enables phased deployment of applications from an Internet-based environment to a physical deployment, i.e. application code can be tested first on a virtual sensor node executing on PC's and/or PDA's provided that the same API was preserved on in-situ micro sensor nodes. For example, the MOS user-level network stack permits a network layer routing algorithm to be tested first on virtual sensor nodes on a Linux PC before being deployed. The emStar system also advocates cross-platform support though the approach is focused on TinyOS, as explained later [6].

As added benefits to this cross-platform approach, MOS achieves code reuse and a low barrier to entry in terms of programming for sensor networks. For example, a standard stop-and-wait reliable protocol as well as a standard RC5 security algorithm [10] are both available as C code, and have been ported into MOS. Also, the standard programming language and standard threading model ease the barrier to entry to programming for sensor networks. Since the kernel is also written in C, then kernel development can leverage the same skills used for application development.

MOS is also designed to provide advanced remote management capabilities for in-situ sensor networks. Towards this end, the goals of MOS are to support useful yet sophisticated features, including dynamic reprogramming of sensor nodes via wireless, remote debugging of sensor nodes, and multimodal prototyping of virtual and deployed sensor nodes.

In the remainder of the paper, Section 2 describes the MOS architecture, including scheduler and network stack, and how MOS achieves a lightweight implementation. Section 3 provides a discussion of different sensor OS models and programming paradigms. Section 4 describes how the multithreaded MOS achieves power efficiency. Section 5 explains the goals of MOS with respect to in-situ features. Section 6 summarizes the MANTIS hardware nymph. Section 7 concludes with future work.

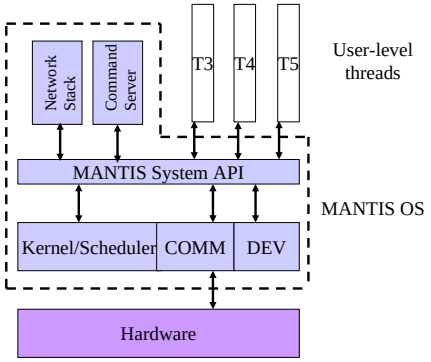

Figure 1: MANTIS OS architecture compresses a classic multithreaded layered operating system design into $<$ 500 bytes of RAM.

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. Application threads are separated by the API from the underlying OS. By preserving the API across platforms, MOS enables cross-platform support. MOS consists of a lightweight and energy-efficient scheduler, a user-level network stack, as well as other components such as device drivers.

2.1 Applications and APIs

MANTIS provides a convenient environment for creating WSN applications. Figure 2 illustrates a simple yet commonly used **sense_and_forwa** application thread, which is available along with the complete open source MANTIS software release at http://mantis.cs.colorado.edu/. This simple application thread, which runs on a micro sensor node such as the MICA2 mote, reads a sensor value from an analog to digital converter (ADC) port, toggles the LED, and then transmits the value of the sensor over the radio - all in about ten lines of code.

All applications begin with **start**, which is similar to main(). The system properly initializes other system-level threads, such as the network stack, which is just another application thread. Though not

```
#include <inttypes.h>
#include "led.h"
#include "dev.h"
#include "com.h"

void start(void)
{
    uint16_t delay;
    uint8_t value;
    comBuf send_pkt;

    value = dev_get(DEV_MICA2_LIGHT);
    mos_led_toggle(0);

    send_pkt.data[0] = value;
    send_pkt.size=1;
    com_send(IFACE_RADIO, &send_pkt);
}
```

Figure 2: Simple sample code of sense-and-forward application sender.

shown in this example, this sense-and-forward application thread can spawn new threads by calling **thread_new**, as can all applications.

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 [11] 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.

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*: **com_send**, **com_recv**, **com_ioct**, **com_mode**
- *On board sensors (ADC)*: **dev_write**, **dev_read**
- *Visual Feedback (LEDs)*: **mos_led_toggle**
- *Scheduler*: **thread_new** (could have been used, but was not)

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 [12] can be compiled to run on X86 PCs with little to no alteration.

2.2 Kernel and Scheduler

The design of the MOS kernel resembles classical, UNIX-style schedulers. The services provided are a subset of POSIX threads [15], 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 encouraged to dynamically allocate heap space, although that was an API decision and is not an inherent limitation of MOS. This limitation is imposed because with such limited memory it is important to have a well planned and coherent memory management policy.

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 scheduler 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. The time slice is configurable, and is currently set to about 10 ms.

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.

2.3 Network Stack and "Comm" Layer

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, as shown in Figure 1, following the design of ALPINE [16]. A user-level network stack enables easy experimentation with the network stack in user space, and also enables cross-platform prototyping of network stack func-

tionality on X86 PCs prior to deployment in WSNs, e.g. a new data-driven routing protocol can be tested in virtual sensor nodes on Linux PCs before deployment, as explained in a later section. The term user space is more aptly applied to manipulation of the network stack on X86 PCs, where there is a clear distinction between user space and kernel space, rather than on ATMega128 MCU sensor nodes, where there is no such distinction.

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. In this way, the network stack avoids data copies and resembles the zero copy approach of TinyOS, SMAC [17] and zero copy sockets [18].

The stack supports layer three and above, i.e. network layer routing, transport layer, and application layer. MAC protocol support is performed by the communication layer, also called the "comm" layer, which is located in a separate lower layer of the OS, distinct from and below the user-level networking stack.

The MOS comm layer provides a unified interface for communications device drivers (for interfaces such as serial, USB, or radio devices). The comm layer is shown in Figure 3. The comm layer also manages packet buffering and synchronization functions. The network or application thread interacts with communications devices through four functions: *com_send, com_recv, com_mode, and com_ioctl*.

When com_send is called, the sending thread (the network, or perhaps an application thread) passes a pointer to a packet buffer, called a comBuf. The comm layer blocks the sending thread and passes the pointer to the specified device driver. While device drivers may be implemented as threads, the typical implementation is in terms of an interupt-driven state machine. This state machine proceeds to send the packet through the hardware device, and the sending thread is resumed when the state machine reaches its complete state.

While sending can be synchronous, receiving must happen in the background even when a network or application thread is not currently making a com_recv call. Memory for received packets is thus managed by the comm layer itself, which owns a number of comBufs. Device drivers may request comBufs, which are then allocated to that device. Once a comBuf is obtained, the device driver may fill it with a received packet, as directed by its interrupt state machine. When a packet reception is complete, the device driver calls com_swap_bufs, which exchanges the full comBuf for an empty one. Full packets are buffered in order by the comm layer. When a thread calls com_recv, it is blocked until a full comBuf on the specified device is available, at which time a pointer to that comBuf is returned. Since the receiving thread now possesses a buffer that was allocated by the comm layer, it must call com_free_buf when it is finished with the buffer; this advises the comm layer that the buffer may be reused. The extra call to free a buffer is more complexity for the receiving thread, but it allows the comm layer to provide true zero-copy service. Also, because the comm layer is completely interrupt-driven, the comm layer achieves zero polling, which is energy efficient.

Besides send and receive functions, the comm layer provides mode and ioctl calls. The mode call is used to power up or power down the device when needed. The meaning of the ioctl call is device-specific.

The MAC layer protocol is located within the device driver for the radio, which is housed in the comm 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, and transmit power control. The MAC layer 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, with a maximum of 64 bytes. An early version of the MAC protocol supported random backoff, while the current version of the MAC supports TDMA for star topologies. MOS will support CSMA in the near future, by adopting and augmenting SMAC and/or BMAC.

The lower layers of the network stack, including MAC and physical layers, occupy about 64 bytes of RAM total to support three communication interfaces, namely the radio, serial link, and loopback interfaces. Additional RAM buffers must be allocated to store packet data. Thus comm buffs are allocated at 64 bytes per buffer, with an added three bytes of overhead per buffer. Currently, five such buffers are allocated, though the plan is to allocate more buffers from all RAM. Since these buffers will be passed directly to applications, then there are zero copies. An additional small amount of space is 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 size of the user-level network stack will depend on the complexity of the protocol(s) the user desires for implementation. Overall, the network stack consumes less than 200 bytes of RAM.

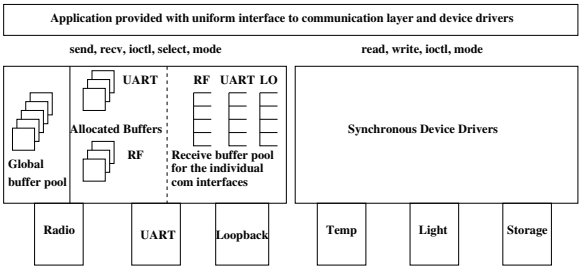

Figure 3: The MOS communication COMM layer (left) is designed for asynchronous I/O with the radio, serial, ... and achieves zero copy operation and zero polling. The device DEV layer (right) is designed for synchronous I/O, e.g. sensor readings.

2.4 Device Drivers

MANTIS adopts the traditional logical/physical partitioning with respect to device driver design for the hardware. The 'device layer, or "dev" layer shown in Figure 3 houses drivers for synchronous I/O, e.g. sensors, external storage, etc. Drivers for asynchronous communication, e.g. radio or serial, are housed in the comm layer. Several POSIX-style system calls *dev_read(), dev_write(), dev_mode(), and dev_ioctl()* are implemented for each device in a simple device layer. A single static table is used to store function pointers for each device's implementation of the device layer model. Devices are specified by their index into this table rather than using a file descriptor, to save on code size and memory usage. Since the table is static, there is some lost overhead if it is not full. Each device has only 4 functions to implement, and a mutex, so this overhead is minimal. After the initialization of the device, a call to dev_register() is made, to place the device's function pointers into the call table, and initialize the mutex associated with the device. This driver scheme has been implemented for EEPROM, several

assorted sensors, and is planned for accessing flash storage. We foresee drivers for each possible device easily conforming to this model.

The dev_mode() call is provided to interface with the power management system. Devices can currently be in a state of on, off, or idle. If users know they will not be using the device for a period of time, they may set it to off or idle, depending on their needs, for a savings on power consumption. Before accessing the device again, its mode must be set back to on.

The dev_ioctl() call is a generic function, taking device-specific variable arguments. A device such as an EEPROM can use this function to set the memory address where dev_read() and dev_write() access the hardware. As more drivers are written, device-specific routines that do not fit into the normal device model will be accessed through this interface.

2.5 Summary

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 ATMega128's 4 KB of RAM, as well as sufficient storage in the ATMega128's 128 KB of flash storage.

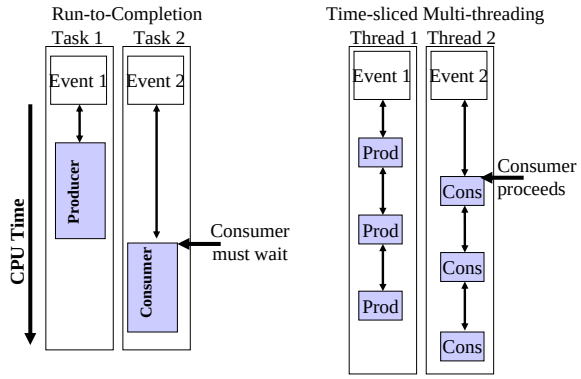

Figure 4: A Bounded Buffer Producer/Consumer Application avoids buffer overflow in a preemptively time-sliced system of interleaved threads (right), because the consumer can execute and empty the buffer earlier. A single-threaded event-driven run-to-completion system (left) can overflow the buffer when a consumer task is forced to wait for a long-lived task to finish, e.g. producer or other complex task.

3. DISCUSSION: THREADS AND EVENTS

This section discusses first the benefits and then the costs of preemptively time-sliced multithreading in sensor systems. The following discussion refers to Figure 4, which depicts two execution models for sensor systems: an event-driven run-to-completion single-threaded approach on the left; and a preemptively time-sliced multithreaded model on the right. The literature contains a variety of examples of thread-driven and event-driven systems, as well as comparisons between the two models [55, 56, 57, 58, 59, 3]. A recent paper suggests that thread-driven systems can achieve the high performance of event-based systems for concurrency intensive applications, with appropriate modifications to the threading packages [55].

TinyOS is a standard embedded OS for sensor networks based on an event-driven design philosophy. The simplicity of the system is tailored for event-driven sensor I/O. Tasks run to completion with respect to other tasks but may be interrupted by events. Only one stack is needed because only one task is running, so that there are no context switches. TinyOS also assumes a modularized programming language nesC, an extension of C. It is a static language in which all run-time memory usage is preallocated. Besides its modularized fashion, nesC also analyzes the code and handles concurrency inside the language instead of at the user level. The designers of TinyOS also believe that an event-based approach is able to create an energy-efficient system since there is neither blocking nor polling in an event system. Unused CPU cycles can be spent in the sleep state as opposed to actively looking for an interesting event. TinyOS provides an event-driven paradigm that meets the requirements of simple tasks characteristic of today's sensor nodes.

The MANTIS multithreaded OS seeks to provide a pathway to evolve sensor systems to support increasingly complex tasks, while at the same time meeting the resource constraints of energy and memory typical of sensor networks. Time-sliced multithreading offers automatic preemption, which has the advantage that a single segment of application code cannot block the execution of other tasks. This is important in sensor systems, since blocking certain time-critical tasks from executing, such as network packet processing, can result in overflow of network buffers when tasks are sufficiently long-lived and a sensor node's RAM buffers are sufficiently small.

To illustrate the bounded buffer producer-consumer problem as it applies to sensor networks, Figure 4 depicts two tasks, namely a producer and a consumer. Such pairs of tasks are common, and typically share a buffer between them, not shown. As the producer generates data, the producer places this data in the bounded buffer between the two tasks. The consumer empties the buffer whenever it has a chance to execute. If the consumer is unable to execute for some time while the producer continues to add data to the buffer, then the buffer will eventually overflow.

For sensor networks, a typical consumer would be a network stack that needs to process incoming packets and route these packets to/from the radio. In a typical sensor network topology, a sensor node relays data from several children nodes to a parent node that is closer to the ultimate destination, namely the base station. An arriving radio packet typically causes an interrupt that can preempt an executing task. A small interrupt handler then transfers the packet to a buffer for complete processing at some later time by another task such as the network stack. Thus, multiple downstream nodes act as producers of sensor data whose packets are received and deposited into the relay node's buffer, awaiting further processing by the relay's network stack.

As sensor nodes are called upon to perform increasingly complex tasks, the likelihood increases that a long-lived task in a run-to-completion system can block processing by the network stack consumer, resulting in lost packets due to buffer overflow. Such tasks could include aggregation via standard compression algorithms, standard encryption/decryption, and standard signal processing techniques. Though today's aggregators typically do little more than summarize multiple sensor values by calculating an average, it is not unreasonable to expect aggregation to employ more sophisticated yet memory-efficient compression algorithms in the near future. For example, we have ported a lightweight compression algorithm that uses arithmetic coding to the MICA2 motes. This compression code executes in the 4 KB of RAM provided by the ATMEL chip. Other researchers have ported the LZ77 compression algorithm to the MICA2 [48]. Similarly, though today's sensor nodes employ scalar values to trigger actions such as routing, e.g.

temperature $> T$ degrees, future behavior may well be triggered by simple frequency analysis of the sensor data, e.g. the sensor data has a strong tone at 1 kHz. Thus, we have also ported a standard 64-point FFT algorithm based on integer arithmetic for fast execution on the MICA2 motes. Other researchers have implemented a much slower floating point FFT on the MICA2 [48]. In previous work, we have implemented both RC5 and AES encryption on MICA motes [29], but present improved implementations in this paper. The resulting execution times are summarized in Table 1.

Table 1: Execution times for various complex tasks on MICA-2

Complex Task	Execution Time
Arithmetic Coding: Compression	321 ms
Arithmetic Coding: Decompression	504 ms
64-point FFT (integer)	56 ms
512-point FFT (floating point)	30 sec [48]
RC5 encryption (12 rounds)	2 ms/32 bytes
AES encryption (no pre-computed table)	28 ms/32 bytes

We found that that a standard compression task such as arithmetic coding is relatively long-lived. Arithmetic compression of 128 samples of data took 321 ms, while arithmetic decompression of 128 samples took 504 ms. Arithmetic coding was chosen for its near-optimal compression efficiency. An alternative would be Huffman coding, which would sacrifice compression for a speed improvement of about a factor of two [19]. Also, the 64-point integer FFT required 56 ms to execute. We expect that 128-point and 256-point FFTs will require hundreds of milliseconds to run to completion. AES, in low memory mode, required 28 ms to complete encryption on just 32 bytes of data.

Any task that is sufficiently long-lived during its run to completion is a candidate for causing buffer overflow, and lost packets. Suppose there are 200 bytes of RAM devoted to the network stack's buffer. Suppose also that packets arrive from four downstream neighbors at the rate of 5 packets/sec, with each packet of size 30 bytes. In this case, sensor data arrives at a rate of 600 bytes/sec. If any of the above complex tasks takes more than a few hundred milliseconds to run to completion, then the bounded buffer will overflow. In contrast, in a multithreaded system, a network stack consumer thread will be given its time slice despite the presence of long-lived threads and thus be able to process its traffic and limit the loss of data.

To address this bounded buffer problem, a programmer in a run-to-completion system is therefore burdened with several difficult and time-consuming tasks. First, the programmer must decompose their code into sufficiently small execution modules. In some cases, e.g. arithmetic compression, correct partitioning of the code may require a semantic understanding of the algorithm, which is a stiff requirement for a programmer who only wishes to port the code to a micro sensor platform. For example, the decomposition of LZ77 compression code into TinyOS modules required detailed semantic awareness [48]. Second, understanding when the code modules are "sufficiently" small to avoid blocking other tasks depends both upon the other tasks that will be executing as well as their application tolerances, neither of which are known a priori by the programmer. The designer may be forced to choose the finest granularity to avoid the pitfalls of run-to-completion. The least difficult option of porting code as one monolithic module runs the risk of the run-to-completion bounded buffer problem. Third, the programmer must invest significant time to make sure to relinquish control properly in each module. For example, the programmer must ensure that each module avoids busy-wait polling and/or infinite loops in order to avoid indefinitely blocking other tasks.

In contrast, programmers in MOS do not have to alter their programming practices to accommodate the above concerns. A program can be written without having to physically partition the thread of execution, though the programmer is free to generate multiple threads if *desired*. In addition, since there is a vast body of code already written for threaded operating systems, a programmer can port code to MOS with relative ease without requiring a deep semantic understanding of the coded algorithm. A programmer can also write a long-lived task without explicitly ensuring that the program does not block or hinder other processes.

Support for multithreading in MOS comes at the cost of context switching overhead and additional stack memory for each thread. First, our claim is that the context switching overhead is only a moderate issue in WSNs. Each context switch incurs only about 60 microseconds of overhead (about 120 instructions, or approximately 400 clock cycles), since 30 registers need to be reset. In comparison, the default time slice is much larger at 10 ms. This is less than 1 percent of the microcontroller's cycles. Since WSNs are largely focused on I/O, e.g. sensor data acquisition and packet forwarding, and not with the computational performance of compute-bound threads, then the modest slowdown in pure computational speed should not affect the primary function of micro sensor nodes.

The second cost of multithreading is memory allocated for each thread's stack, though this can be mitigated with intelligent stack analysis. The default thread stack size in MOS is 128 bytes. Since 4 KB of RAM are available in the MICA2 motes, and the kernel occupies less than 500 bytes, then there is considerable space left to parameterize MOS to support up to a double digit number of user threads. For the early Rene motes with only 512 bytes total of RAM, it would be infeasible to fit both the MOS kernel and user threads into such a constrained space. Since current and future sensor nodes are likely to contain at least the MICA2's present capacity of 4 KB RAM, then MOS demonstrates that there is sufficient memory to fully support both the OS and multithreaded applications. However, the degree of multithreading remains an issue, because of the possibility of stack overflow. If insufficient space is allocated, then the thread's stack can overflow. To mitigate this issue, we are currently developing stack analysis tools that accurately forecast the stack needs of each thread, and allocate sufficient memory to avoid stack overflow.

The programming paradigm of MOS is based on a standard programming language C. This enables a shallow learning curve, cross-platform support, and code reuse. While nesC is an extension of C, additional investment is required to understand how to program using nesC modules.

Over time, the two types of sensor systems may very well converge and/or coexist. In the future, we could envision a sensor system that combines the best of both the thread-driven system's flexibility as well as the event driven system's efficiency. A thread-driven model provides a general solution for synchronous code, preemption syntax and priority mechanisms. Yet events are well-adapted to many sensing applications as well. By analogy, arguments between the adaptability and reconfigurability of micro-kernels and the performance of monolithic kernels resulted in the development of modular kernels that combined the advantages of both. Moreover, as sensor networks diversify, some micro sensor nodes may run event-driven code and communicate with others that execute via multithreading, e.g. aggregator or application-specific nodes.

```
void start(void){

  Packet data;
  uint8_t i, size;

  mos_enable_power_mgt();

  while(1){
    size = flooding_recv((char *)&data, 0x02);
    if(size < 1)
      mos_led_toggle(0);
    else{
      for(i=0; i<size; i++)
      mos_uart_send(0, ((char *)&data)[i]);
      mos_led_toggle(2);
    }

    mos_thread_sleep(32);
  }
}
```

Figure 5: MOS provides application threads with a sleep(PERIOD) function. If all threads call sleep(), then the OS shuts down the microcontroller until the first sleep deadline expires.

4. POWER MANAGEMENT

A challenge in the design of energy-efficient thread-driven systems is sleeping the scheduler when there are no more threads that need to be scheduled, i.e. all threads are either blocked on I/O, blocked for other reasons or have no useful work to perform. In this section, we describe how MANTIS OS achieves energy efficiency via a sleep() function that is designed to resemble the semantics of the UNIX sleep() function, i.e. taking a parameter for the duration of the sleep, but differs in the behavior of the OS after all application threads have called sleep. This sleep() function enables a threaded system to shut down when there is no meaningful work to do, thereby avoiding energy-consuming busy-wait polling.

A typical wireless node will last only a few days on two AA batteries when used for continuous monitoring. Two AA batteries at 3000 milliampere per hour (mAhr) will last approximately 5 days at 25 mA of power consumption (3000mAhr/25mA = 120 hr or 5 days). The most effective technique for extending the lifetime of in-situ sensor nodes is a low duty cycle that sleeps the node most of the time. Traditional power management techniques for laptops transition between idle and active modes of operation, which is the approach explored in this work. Additional low power methods such as throttling the performance of a processor or turning off only parts of a processor [52], as well as varying the voltage in real-time embedded systems [54] are not pursued in this work.

If sensor nodes are to sleep with a low duty cycle, then the value of the duty cycle must be determined, as well as its periodicity. These important parameters controlling energy efficiency should be both application-specific and adaptive. If a sensor node employs only one sensor, e.g. to monitor temperature once per second, then the period is simply set to one second. However, most sensor nodes have the capability to monitor more than one sensing domain simultaneously, e.g. both temperature and relative humidity [26]. Given multimodal sensing, the low duty cycle behavior becomes more complex. If the temperature sensor is monitored every three seconds, and humidity is monitored every seven seconds, then we desire a sensor OS capable of integrating such staggered and offset application-specific sleep periods. Moreover, we desire that a sensor OS be responsive to changing environmental conditions within the sensing zone. For example, at run time, an application that is tracking an event may wish to change its sampling frequency or periodicity in response to sensed data, e.g. motion of an animal. The sensor OS should provide energy-efficient mechanisms for adapting to run-time changes in sampling frequencies and duty cycles for each application.

Table 2: Power Consumption for MOS in sleep mode and awake mode on MICA-2

	Power
sleep	20 μA
sense and forward (awake)	20 mA
1.0% Duty Cycle for a 300 sec Cycle	66 mAsec
0.5% Duty Cycle for a 300 sec Cycle	36 mAsec

In addition, the behavior of compute-bound applications such as aggregation is far different than data-driven I/O-bound tasks such as periodic temperature and humidity sensing. An aggregation application may wish to delay sleeping of a sensor node until its computation is complete, regardless of the various sensing periodicities. A sensor OS should therefore be flexible enough to accommodate compute-bound application behavior in addition to data-driven sensing applications with varying periodicities and duty cycles.

These examples illustrate that a sensor OS should provide application-specific mechanisms that enable diverse applications to indicate when and how often they wish to sleep, in order to achieve power efficiency. The sensor OS should combine these application-specific natures and emerge with a scheduling timetable that meets application needs while also achieving energy efficiency. The OS scheduler should determine when it is safe for the system to sleep. In addition, the sensor OS should adapt to changing conditions, so that applications at run time can change their sleep times, patterns and periodicities.

As an initial step towards building sleep-oriented capabilities, we have implemented a sleep() function as shown in Figure 5, similar to the UNIX sleep() function. First, the application thread must enable power-save mode, which is accomplished by the call *mos_enable_power_mgt()*. All threads should enable power-save mode, though by default this option is not turned on and must be explicitly activated. This was chosen to maintain compatibility with the UNIX sleep()'s behavior. Next, the application thread may call *mos_thread_sleep(PERIOD)*, with a parameter PERIOD specifying the duration to sleep. This was chosen to mimic the behavior of UNIX sleep(). However, MOS adds the capability that, if all application threads call sleep, then the system truly sleeps most of the time. For example, if there is one application thread, then the system will periodically wake up according to the PERIOD specified by that thread. If there are multiple application threads each calling sleep(), and each specifying a different sleep duration, then the scheduler will keep track of when the earliest deadline expires and wake the system; otherwise, the system will sleep.

Table 2 lists the current consumed by a sense and forward application thread on a MICA2 mote running MOS for different duty cycles. While actively executing the application code, the MICA2 mote running sense and forward consumed 20 mA. However, while the application thread slept, the power consumption was only 20 μA. This confirmed that MOS was correctly sleeping the microcontroller and was also able to periodically wake the thread to execute its code. Thus MOS is able to achieve energy efficiency while maintaining a threaded scheduling capability.

Figure 6 illustrates the energy-efficient MOS scheduler. The ready Queue of the MOS scheduler consists of five priorities, high to low: Kernel, Sleep, High, Normal and Idle. The scheduler se-

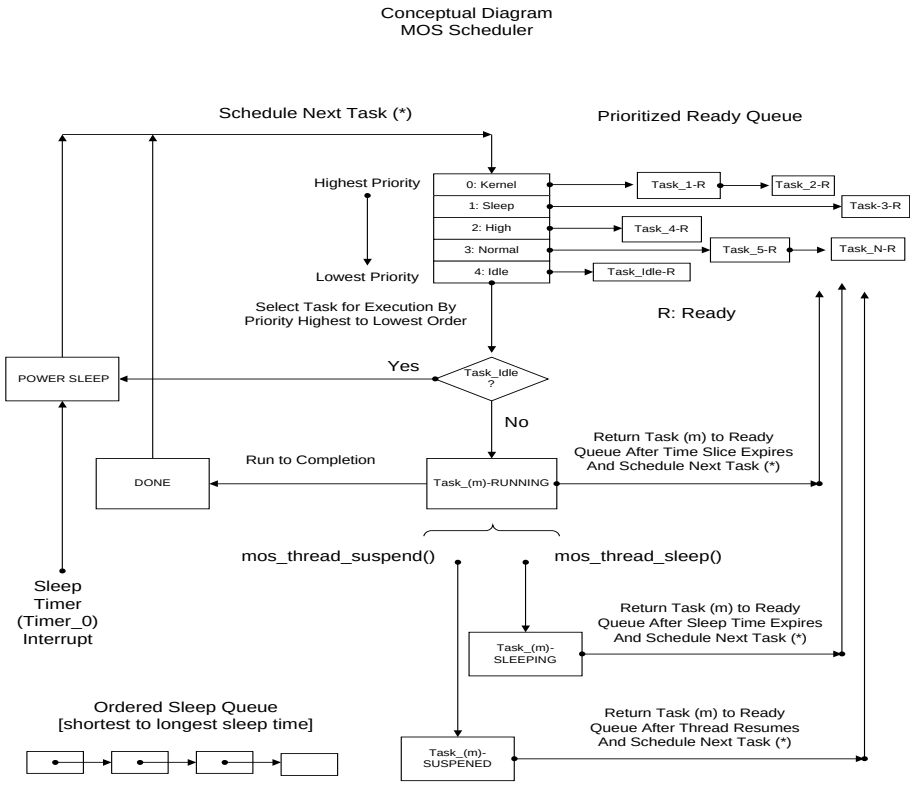

Figure 6: Energy-efficient MOS scheduler.

lects the highest priority ready task and either executes it to completion or puts it in the ready queue if its time slice expired. The scheduler uses 16-bit Timer1 for for multi threading and time slicing. When no threads are ready for execution, then the system sleeps automatically instead of spinning at a 100in the idle loop. The depth of sleep varies as follows. If the system is suspended on I/O, then the system enters the ATMEL's moderate idle sleep mode. Otherwise, if all application threads have called sleep(), then the system enters the deep power-save sleep mode. A separate sleep queue maintains an ordered list of threads that have called sleep(), and is ordered by sleep times, low to high. When the sleep time of the thread in the front of the queue expires, the queue is shortened and sleep times are readjusted. Timer/Counter0 is used to implement the sleep timer because it allows clocking from an external 32 kHz Watch Crystal independent of the internal CPU clock, which is necessary to wake the processor from deep sleep. The *sleep* priority in the ready queue enables newly woken threads to have higher priority so that they can be serviced first after wake up.

MOS also achieves energy efficiency by implementing the comm layer so that it is completely interrupt-driven. There is zero polling in the comm layer.

5. ADVANCED FEATURES OF MANTIS OS

Sensor networks impose additional unique demands on the design of operating systems beyond 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.

5.1 Bridging the Internet and Sensor Network with Multimodal Prototyping

The MANTIS prototyping environment provides a framework for prototyping diverse applications and bridging these applications between the Internet and the deployed sensor network. 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 prop-

erty 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 7. 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 modes 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 renderering 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. For example, our "cortex" visualization application connects to two API's: the MANTIS API in order to behave as a virtual sensor node and receive sensor data streams; and a second graphical API in order to render sensor data. This flexibility is illustrated in Figure 7.

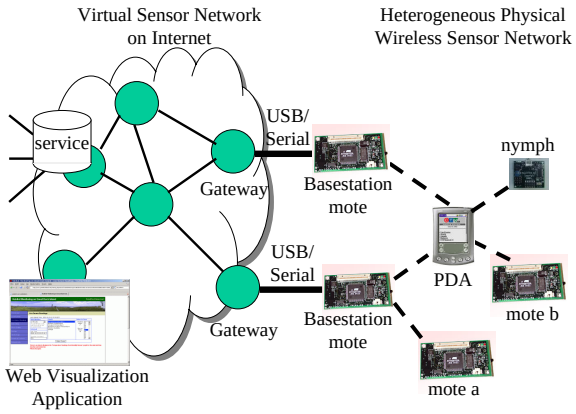

Figure 7: Virtual MOS sensor nodes on the Internet are seamlessly connected with real MOS sensor nodes by preserving a common cross-platform API across X86 PCs and ATMEL micro sensor nodes.

MANTIS achieves a multimodal prototyping environment by preserving a common C API across all platforms. This approach resembles WINE [20], but eliminates the problems of hidden system calls, since all such calls are publicly known in MANTIS. Due to the wide availability and support by the GNU tool chain for multiple platforms, 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 8 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 7 shows the structure of the network, with the two networks connected to each other via a serial RS232 link. Thus, a com_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 com_send(..) call. A com_send() call on the XMOS nodes causes the data to be transmitted over the IP network instead.

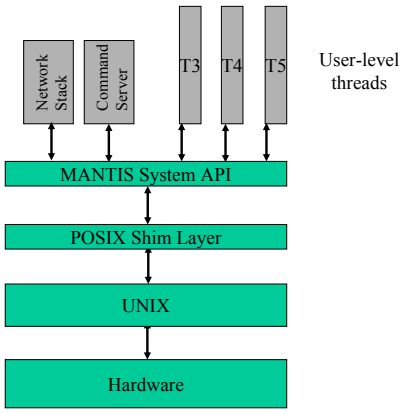

Figure 8: x86 MANTIS OS (XMOS) architecture uses the POSIX shim layer to translate to/from underlying OS.

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 [21], and still others may coexist to perform multicast routing [22]. 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 [23], 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 [24]. Sensorsim is an extension to ns2 and provides a simulation framework that models the sensor nodes and also provides a hybrid simulation combining the real and virtual network [25]. 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 for wireless sensor networks that shares many of the principles of the MOS system. emStar combines pure simulation, hybrid mode and real distributed deployments [7]. Just as in MOS, the same code can be reused in the simulation environment and the real platform, whose targets include the iPAQ and Crossbow Stargate platforms. Just as in MOS, a POSIX compatible programming interface is provided. TinyOS is supported by the emStar framework.

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.

5.2 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 [26]. Since sensor networks may be deployed in inaccessible areas and may scale to thousands of nodes [27], 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.

The goal of MOS is to achieve 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. At the present time, MOS can support remote login and changing of variables/parameters.

Support for dynamic reprogramming of the entire OS is in progress. 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 external storage (flash or 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, the intent is for an application–such as the MANTIS command server–to 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.

Reflashing parts of the OS, e.g. one thread, is a difficult research challenge that will be addressed after dynamic reprogramming of the full OS image has been completed.

Current solutions for dynamic reprogramming [28] 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 micro sensor node.

5.3 Remote Shell, Cortex Application and Command Server

Existing solutions for monitoring sensor networks consider topology extraction [36] and computing summaries of network properties for energy efficient monitoring of sensor networks [37]. 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.

The MCS itself is implemented as an application thread. It listens on the serial and radio for commands either sent to the kernel or to an application. The user may view the list of functions supported by the MCS, inspect and modify the node's memory, change configuration settings, run or kill programs, view the thread table or restart the node. Additionally user applications can register their own functions to be called when a specific command is entered from the shell. After this function is called, the user application can receive parameters for their function. This allows user applications to remain dormant until a command is issued. The shell is a powerful debugging tool, since it allows the user to examine and modify the state of any node, without requiring physical access to the node.

The remote shell is part of a visualization application called the "cortex" that runs on a remote laptop. Figure 9 illustrates an example of the cortex visualization GUI that renders and plots sensor

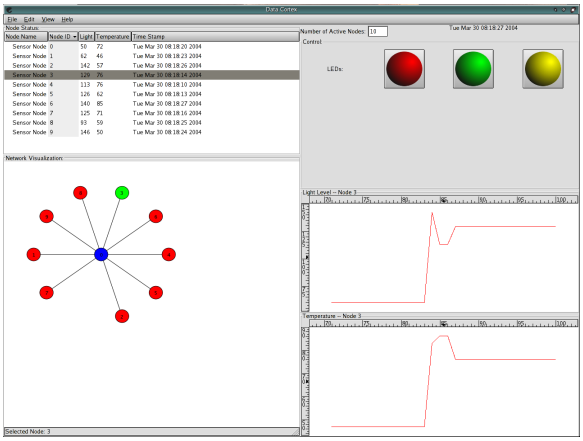

Figure 9: Screenshot of the cortex visualization application for rendering sensor data, part of the MANTIS release.

data in real time, maintaining multiple sliding window histories. This application is included as part of the MANTIS release. The cortex can be used not only to receive sensor data, but also to initiate commands to the sensor network. In this demo application, clicking on the LED's will light the LED on all sensor nodes. More advanced commands are sent via the remote shell interface, not shown in this screenshot. The cortex application is an example of an XMOS virtual sensor node. More precisely, a server application acts as an XMOS virtual sensor node, receiving sensor data packets from the real sensor network. The visualization GUI then connects to the server. This concept of connecting Internet applications as part of the virtual sensor network in order to receive data is illustrated in Figure 7.

6. MANTIS HARDWARE

The MANTIS hardware nymph's design was inspired by the Berkeley MICA and MICA2 Mote architecture [3]. 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 10, altogether incorporating a low power Atmel Atmega128(L) microcontroller (MCU) [12], analog sensor and digital ports, a low power Chipcon CC1000 multi-channel RF radio [38], EEPROM, power ADC sensor, and serial ports on a quad-layer $3.67 * 3.3$ 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 the figure. 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 10. 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.

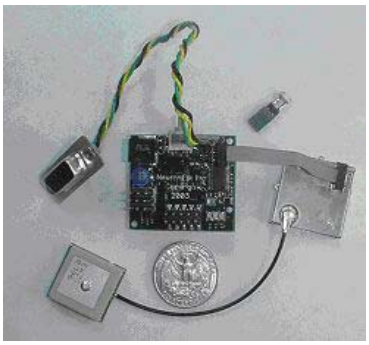

Figure 10: MANTIS nymph micro sensor node.

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 advanced add-on boards with mating connector. This connector has direct access to the MCUs external interrupt pins, I^2C bus, data lines, timers, and pulse width modulation (PWM) pins. Some potential add-on boards would be I^2C expanders that use the interrupt and I^2C 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^2C 16-bit ADC. This ADC enables monitoring of the battery voltage level.

Power consumption numbers for the nymph are given in previous work [5]. GPS was found to consume significant power and will require careful power management to limit its impact on battery lifetime. Comparable recent hardware technology with GPS capability includes the MICA2 Motes [39] and the GPS-enabled GNOMES [40].

7. FUTURE WORK

The MANTIS system is still very much a work in progress. Low power management continues to be a challenge, though sleeping the scheduler has largely eliminated the wasteful busy waiting or polling of normal threaded systems. A follow-on approach would incorporate dynamic hints from within the application with a **power_hint** call to modify the applications requirements dynamically. Prior work on power-efficient scheduling and systems should be leveraged [49],[50]. 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 [51].

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 the 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_recv**(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.

As MANTIS matures, we see several directions to evolve the device interface. For example, with the addition of timers, the devices will gain the ability to set a read interval for multi-byte reads. More specifically, if the user were trying to obtain light samples from a sensor board, currently they are only able to read one byte at a time. With the addition of timers, users will be able to set the read interval through a dev_ioctl() call, and their dev_read() call, called with a multi-byte size, will fill in the buffer of that length, one byte at a time, for each interval. As this operation will block, this provides an ideal method for filling in a radio packet with sensor values over a period of time, and sending only full radio packets as enough data are received.

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 enables medium contention among multiple virtual nodes.

The MANTIS project was awarded an NSF SENSORS 2003 grant to study the role of sensor networks in fighting forest fires Stay tuned to the MANTIS Web site http://mantis.cs.colorado.edu

8. CONCLUSION

The MANTIS sensor system achieves a lightweight classically structured multithreaded operating system in a memory footprint of less than five hundred bytes, including scheduler and network stack. The MANTIS OS achieves energy efficiency by implementing a sleep function. Its power-efficient scheduler recognizes when all threads are sleeping and then sleeps the microcontroller for a duration deduced from each thread's sleep time. MOS supports a simple C API that enables cross-platform support, reuse of a large installed code base, and a low barrier to entry in terms of programming for sensor networks. MOS also supports advanced sensor OS features such as multimodal prototyping, dynamic reprogramming, and remote shells. The MANTIS nymph offers a single-board GPS-enabled solution that is also extensible.

9. REFERENCES

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, "A Survey on Sensor Networks", IEEE Communications Magazine, August 2002

[2] J. Kumagi, "The Secret Life of Birds", IEEE Spectrum, April 2004, vol. 41, issue 4, pp. 42-49.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister "System Architecture Directions for Networked Sensors". Proceedings of Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), November 2000.

[4] 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. 581590, Oct 2000.

[5] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, R. Han, "MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors", 2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA) 2003, pp. 50-59.

[6] 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.

[7] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, D. Estrin, "EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks", to appear in the Proceedings of USENIX 04.

[8] F. Martin, B. Mikhak, and B. Silverman, "MetaCricket: A Designer's Kit For Making Computational Devices," IBM Systems Journal, vol. 39, nos. 3 and 4, 2000.

[9] Crossbow motes, http://www.xbow.com.

[10] R. L. Rivest. "The RC5 Encryption Algorithm",Proceedings of the 1994 Leuven Workshop on Fast Software Encryption, pages 86-96

[11] 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.

[12] Atmel AVR 8-bit RISC processor, http://www.atmel.com/products/AVR

[13] AVRX Real-Time Multitasking Kernel for the Atmel AVR series of micro controllers, http://www.barello.net/avrx/index.htm.

[14] J. Labrosse, MicroC/OS-II: The Real-Time Kernel, 2nd edition, CMP Books, November 1998.

[15] Portable Operating System Interface(POSIX) - Part 1: System Application Programming Interface (API)[C Language] ISO/IEC 9945-1:1996, IEEE

[16] 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.

[17] 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.

[18] H. K. Jerry Chu, "Zero-Copy TCP in Solaris", Proceedings of the USENIX 1996 Annual Technical Conference, San Diego, California, January 1996.

[19] A. Bookstein and S. T. Klein. " Is Huffman coding dead?". Computing, 50:279– 296, 1993.

[20] WINE, http://www.winehq.com/.

[21] C. Intanagonwiwat , R. Govindan , D. Estrin, "Directed Diffusion," ACM MobiCom 2000, pp. 56-67

[22] 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.

[23] P. Levis and N. Lee. "Simulating Tinyos Networks". http://www.cs.berkeley.edu/ pal/research/tossim.html.

[24] L. F. Perrone and D. M. Nicol: A Scalable Simulator for TinyOS Applications, Winter Simulation Conference, 2002.

[25] 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.

[26] 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.

[27] 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.

[28] P. Levis, D. Culler "Mate: a Virtual Machine for Tiny Networked Sensors", ASPLOS, Oct. 2002.

[29] J. Deng, R. Han, S. Mishra, "A Performance Evaluation of Intrusion-Tolerant Routing in Wireless Sensor Networks," IEEE 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03), 2003, Palo Alto, California, pp. 349-364.

[30] 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.

[31] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan, "The Cricket Location-Support System", Proc. of the Sixth Annual ACM International Conference on Mobile Computing and Networking (MOBICOM), August 2000.

[32] M. Leopold, M. B. Dydensborg and P. Bonnet. "Bluetooth and Sensor Networks: A Reality Check". 1st ACM conference on Sensor Systems,(Sensys'03) LA, CA, November 2003

[33] The Smart-Its project, http://www.smart-its.org/.

[34] The Eyes project, http://eyes.eu.org/.

[35] J. D. Case, M. Fedor, M. L. Schostall, and C. Davin. RFC 1157: Simple network management protocol (SNMP). RFC, IETF, May 1990

[36] 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

[37] 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

[38] Single chip ultra low power RF transceiver http://www.chipcon.com/files/CC1000_Data_Sheet_2_1.pdf, 2001

[39] Crossbow, http://www.xbow.com/.

[40] 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.

[41] 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 Zebranet", In ASPLOS, San Jose, CA, October 2002.

[42] 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

[43] J. Elson, L. Girod, D. Estrin "Fine-Grained Network Time Synchronization using Reference Broadcasts", In OSDI 2002, Boston, MA. December 2002.

[44] J. Elson, K. Rmer, "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

[45] H. Dai, R. Han, "TSync : A Lightweight Bidirectional Time Synchronization Service for Wireless Sensor Networks", ACM SIGMOBILE Mobile Computing and Communications Review, Special Issue on Wireless PAN and Sensor Networks, vol. 8, no. 1, January 2004, pp. 125-139.

[46] S. Ganeriwal, R. Kumar, S. Adlakha, M. Srivastava, "Network-wide Time Synchronization in Sensor Networks," Technical report, UCLA, Dept of Electrical Engineering, 2002.

[47] Simple Network Time Protocol, (SNTP) version 4. IETF RFC 2030

[48] Ning Xu, "Implementation of Data Compression and FFT on TinyOS", Embedded Networks Laboratory, Computer Science Dept. USC. Los Angeles, http://enl.usc.edu/ ningxu/papers/lzfft.pdf.

[49] D. Grunwald, C. B. Morrey III, P. Levis, M. Neufeld, K. Farkas, "Policies for Dynamic Clock Scheduling", Operating Systems Design and Implementation 2000.

[50] W. Hamburgen, 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.

[51] 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.

[52] D. Albonesi, R. Balasubramonian, S. Dropsho, S. Dwarkadas, E. Friedman, M. Huang, V. Kursun, G. Magklis,

M. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. Cook, S. Shuster, ”Dynamically Tuning Processor resources with Adaptive processing”, IEEE Computer, December 2003, pp. 49-58.

[53] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, T. Weller, ”Energy Management for Commercial Servers”, IEEE Computer, December 2003, pp. 39-48.

[54] J. Luo and N.K. Jha, ”Battery-Aware Static Scheduling for Distributed Real Time Embedded Systems”, Proc. 38th Design Automation Conference, ACM Press, 2001, pp. 444-449.

[55] R. von Behren, J. Condit, and E. Brewer ”Why Events Are A Bad Idea (for High-concurrency Servers)” 9th Workshop on Hot Topics in Operating Systems (HotOS IX) 2003.

[56] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur ”Cooperative Task Management Without Manual Stack Management”, In Proceedings of the 2002 Usenix ATC, June 2002.

[57] H. C. Lauer and R. M. Needham. ”On the Duality of Operating System Structures”, In Second International Symposium on Operating Systems, IR1A, October 1978.

[58] J. K. Ousterhout. ”Why Threads Are A Bad Idea (for most purposes)”,Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[59] J. Larus and M. Parkes, ”Using Cohort Scheduling to Enhance Server Performance” Technical Report MSR-TR-2001-39, Microsoft Research, March 2001.