Middleware for Long-term Deployment of Delay-tolerant Sensor Networks

Pei Zhang, Christopher M. Sadler and Margaret Martonosi
Department of Electrical Engineering
Princeton University
{peizhang, csadler, mrm}@princeton.edu

ABSTRACT

Wireless sensor networks have a wide range of applications and are deployed in increasingly varied situations. Many deployments have focused on long term monitoring, which uses nodes that are delay-tolerant and depend on low-power sleep to minimize the energy consumption and extend operational lifetime. These nodes often have many software modules contending for system resources, making both software development and power management difficult. These challenges call for middleware layers that are different from those for real-time, dense networks. This middleware must be flexible to accommodate and control the vast variety of available hardware peripherals and software applications, as well as to provide simple methods to manage a node's energy consumption. It must also provide an easy-to-use interface for software modification and at the same time take advantage of the long idle periods typically experienced by delay-tolerant sensor nodes.

The middleware we propose in this paper is designed for long-term use in delay-tolerant networks. Our middleware keeps a unique system time for more than one year, offering very long-term event scheduling. This middleware takes advantage of the low node utilization of long-term networks and executes software modules in sequence. This avoids complexities in context switching of multiple threads on a single threaded processor, and improves the simplicity of software implementation. Our middleware has a small code footprint of less than 3.5KB, as well as very low scheduling overheads of less than 40 μs to run scheduled applications. This structure also allows the system energy to be centrally managed by the middleware, which minimizes node power consumption and simplifies real world software development.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: GeneralHardware/software interfaces; C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*Microprocessor/microcomputer applications*; D.4.7 [**Operating Systems**]: Organization and Design—*Distributed systems*[Real-time systems and embedded systems]

General Terms

Design, Measurement, Performance

Keywords

Sensor Networks, Delay-tolerant Networks, Middleware System, Application Scheduling

MidSens'06, November 27-December 1, 2006 Melbourne, Australia

1. INTRODUCTION

As sensor networks research becomes more focused on long-term deployments with months or years of autonomous operation, many new issues arise. In long term testing and deployments, energy consumption and software updates are major concerns, due to the limited accessibility of nodes. On the other hand, high software concurrency may be a lower priority in these systems since they are delay-tolerant. In particular, while bursts of events sometimes occur, modest queuing delays may be acceptable.

Middleware for current sensor networks, such as TinyOS, is often designed with a dense mesh in mind, where nodes are in close proximity to one another. In these systems, event-processing delay is often a major bottleneck. The middleware must handle multiple events concurrently to reduce this bottleneck and possible data loss.

Long-term sensor networks must have low daily energy consumption. Thus, these systems tend to have low data rates, and are delay-tolerant. Due to the long sleep times and relatively low processing power of the processors used, computation can instead be spaced out over time. These characteristics call for a different approach to middleware systems and how applications are handled.

To address the issues raised by delay-tolerant sensor networks, we have developed a middleware system with the following novel characteristics:

- Single threaded execution allows for more predictable resource usage and less inter-software module interference.
- Time-keeping with unique time for more than one year, allows for very long-term scheduling of applications.
- Application modularity allows for easy code-update.
- Central energy control ensures low energy usage during system idle.

Our scheduling system schedules applications in a non-overlapped manner. Each application can consist of one or

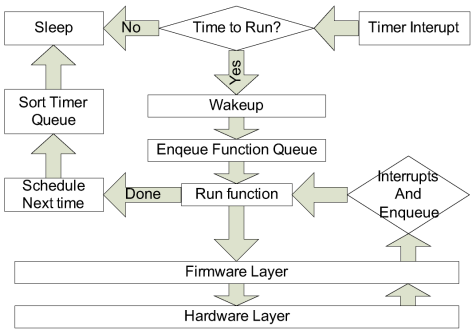

Figure 1: Timer Scheduling Overview

more modules, which are executed in order. This provides an easy interface, and limits the contention problems among multiple software modules, developed by multiple people. The system is controlled by a single application at any one time and thus allows for a more predictable resource usage, and lowers the likelihood of system failure due to unanticipated interference from multiple applications.

This dynamic scheduling system, with unique system time, allows the system to schedule and run applications up to a year in advance. This scheduling also gives the system advance knowledge of system usage, which allows the system to enter and exit low-power idle mode during low activity periods in order to control power consumption. All this is done with an extremely small code memory footprint of 3.5KB and overhead of less than 40μs to execute applications.

Our timer-based middleware provides a platform for easy dynamic scheduling and power management aimed at coarse-grained, long-running sensor network systems. It provides the programmer an easy and intuitive interface to add, remove, and change applications during development or remotely via code updates, which is important since sensor nodes are often deployed in hard to access situations. The system also automatically minimizes energy consumption and self monitors to recover from application failures.

This paper is organized as follow: Section 2 provides the details of our middleware. Section 3 introduce ZebraNet test node, the implementation platform. Section 4 provides example applications. Section 5 shows preliminary measurements for the system. Section 6 compares related work. Section 7 discusses future work, and finally conclusions are given in Section 8.

2. MIDDLEWARE OVERVIEW

Our middleware system provides middleware functions that schedule and execute applications, monitoring functions that perform sanity checks, and low-level interrupt handlers for data buffering. While the limitations of sensor network hardware present challenges for a traditional middleware system, they also provide opportunities for optimization.

Our overall goal is to allow for an abstracted hardware layer, while maintaining some of the advantages of a application-specific program. The middleware's main components are: time-keeping functions, for keeping accurate and unique time; a queue system for executing applications based on the schedule; an application scheduler, for creating the schedule; and data interrupt handlers to provide seamless data reception. These components are discussed in more detail below.

2.1 Timekeeping

Accurate timekeeping is an important aspect of the scheduling system. Its accuracy and simplicity allow a system to spend more time in the idle low-energy state. To keep the system time from overflowing often, and to provide a unique time for advance scheduling, the system time is kept in a 32-bit number. This allows the system to have a unique system time for 388 days, which is long enough that overflow does not occur often in the deployment lifetime. In our implementation, a limit is set at 365 days. If a longer application sleep period is desired, an intermediate function must be scheduled for within 365 days; this function will then schedule the desired function.

The microcontroller used in our implementation offers two clocks modes: a fast 4MHz clock and a slow 4KHz clock [11]. Every 1/128 seconds the system wakes up to an intermediate state running on a low speed 4kHz clock, giving 32 cycles for each interrupt. However the timer interrupts only consists of updating the system clock and checking for scheduled applications. This uses 22 cycles, sufficient to avoid skipping interrupts. If an application is scheduled during this time slot, the system wakes up and runs on the 4MHz clock to execute the application. During the low-energy idle state, the timer continues to run, providing an accurate time. The process is depicted in Figure 1. This multi-stage wake up allows the system more time in idle low-power states.

To avoid problems with time drift, time synchronization is currently done in two ways. One method to synchronize time in our system is through a GPS peripheral. The GPS protocol specifies an extremely accurate clock, which the GPS receiver uses to calculate its position. When the GPS receives a valid position lock, the accurate time from the GPS can be used to update system time. For nodes with a GPS sensor, this method provides a precise system clock. Another method is synchronizing at startup, where a powerful node sends a radio packet to synchronize all the nodes. This method is used mostly for testing purposes. Other time synchronization methods can also be integrated into the system.

2.2 Queue System

The system is run mainly via two components, a timer kernel and a function queue. The timer kernel is used to schedule and invoke applications; the function queue is used to maintain order and run the applications. The queues allow the system to avoid conflicts and use only pointers to applications, allowing easy upgrades and schedule changes.

2.2.1 Timer Kernel

The timer kernel is used mainly for scheduling applications. To run an application at a given system time, the scheduler simply passes the application pointer and the desired invocation time to the kernel, which enqueues the function in a timer queue. A valid variable is also included which allows the system to cancel the run of the scheduled application without changing the contents of the queue. Each time a new item is enqueued, the timer kernel sorts the queue by invocation time. Since sorting delay is incurred at scheduling time, not execution time, short applications can be scheduled in adjacent time slots, within 1/128 second.

The timer queue is checked by the timer kernel every time slot. When the event time is reached, the scheduler transfers the function from the timer queue to the function queue and

wakes the rest of the system. If the desired execution time has passed for this function, due to another active application, the timer will still transfer the function to the function queue to avoid skipping applications. This is rare, because applications are usually well-spaced.

To ensure stability and to make the system self-recoverable, the timer kernel monitors the system's timer queue. If the timer queue is empty during system sleep, or the scheduled events do not include certain crucial applications, the timer kernel would detect a system failure and either restart the system or take other appropriate measures defined by the programmer.

2.2.2 Function Queue

The function queue is a simple queue that contains pointers to functions that are ready to be executed in order. When the system is taken out of the idle state, the function queue is non-empty with at least one entry having been inserted by the timer kernel. The function queue is self-blocking in that functions are executed in order. This limits the ability of the applications to be executed concurrently unless the application is specifically programmed to exit and requeue itself during long delays. This feature reduces the hardware contention of applications and simplifies both application and system software.

2.3 Application Scheduler

The application scheduler determines when an application is run. The scheduler is required by the middleware, and is called at the end of the application or in the initialization routine. The scheduler can be very simple, for example scheduling repeated events at fixed intervals; or very complex, adapting to and accounting for many aspects of the physical state and environment of the node. The use of a scheduler by the system allows for an extremely flexible way of scheduling events that can be easily modified by small remote code updates.

2.4 Hardware Interrupt Handling

While the applications are executed mostly in a single threaded manner, the hardware interrupt events can interrupt the system and receive data without the need to wait for an application to exit. This is done to prevent possible data loss, given the one-byte hardware buffer available on the processor. The interrupts only move data from the hardware buffers to a software buffer in RAM. This adds very little processing time, and since all data interrupts are processed this way, there is minimal chance of hardware conflicts.

2.5 MAC Layer

We provide a simple random back-off MAC layer. When the communication starts, each node sends out a peer packet for mutual discovery. Different nodes contest for the channel and employ a random back off in case of collisions. When the node has no data to send and it is no longer receiving data, it will reach timeout and exit the communication slot to go into idle low-energy state. This method of adaptive MAC layer is similar to T-MAC [12].

The MAC layer is implemented in two locations. Most of the collision avoidance and detection are processed on the radio module. The peer discovery and session information is done by the network application. The network application is described in Section 4.3.

Schedule a function to execute at a specific time:
int Timer_Function_Enqueue(Function Name, Time to Execute)
Returns Timer ID

Schedule a function to execute after a specific amount of time:
int Timer_Function_Enqueue_based_on_offset(Function Name, Delay)
Returns Timer ID

Have the middleware execute a function as soon as possible:
char Function_Queue_Enqueue(Function Name)
Returns 0 on success, -1 on failure

Cancel a scheduled function:
void timer_cancel(Timer ID)

Figure 2: Middleware Queuing API.

3. HARDWARE PLATFORM: THE ZEBRANET NODE

Our middleware is implemented on the ZebraNet v5.1 test nodes. This latest version of the ZebraNet node is designed especially for experimentation and debugging. It is electronically the same as nodes that were embedded into animal collars for the second ZebraNet deployment in Kenya in June 2005.

The test node consists of several independent units: a microcontroller, flash, GPS, radio, battery charger, battery gauge, and a USB unit for debugging. In particular the Radio and GPS are high-power peripherals, which if controlled improperly would leads to short system life. The power consumption of selected peripherals is shown in Table 1.

Table 1: ZebraNet Test Node Component Power measured at 4.1V battery voltage. The (*) indicates a datasheet value is used to calculate the result.

Component	State	Current
Radio	1W Transmit	1.15A
	Receive	100mA
	Power off	$<1\mu$A
GPS	Tracking	21mA
	Power off	$<1\mu$A
Microcontroller	Fast Clock	1.8mA
	Slow Clock	50μA*
	Full Sleep	2.0μA*
Battery Gauge	Active Mode	52μA*
	Sleep Mode	1.0μA*
Total	Sleep Mode	3.0μA*

A unique feature of the ZebraNet board is the battery unit. The battery unit includes a 2A-hour Li-ion Battery with a BQ27200 battery gauge [10]. This gauge allows the system to select energy conservation techniques based on the level of charge.

Similar to other peripherals in the system, the battery gauge also offers sleep modes. For a non-charged system, the device can be shut down when the system enters sleep mode, drawing only 1μA of current. In charged systems, the gauge can enter sleep mode when no charge is applied. While this feature has not been implemented, our middleware system allows this to be easily implemented at the application level.

4. APPLICATION EXAMPLES

The middleware can support flexible and simple application implementations through its queue functions, some of which are shown in Figure 2. In this section we describe three example applications that are used in ZebraNet.

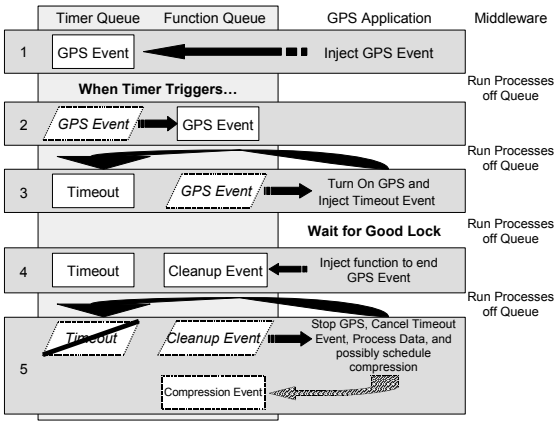

Figure 3: Example of how the GPS application interacts with our middleware. In this example, the GPS is able to acquire a lock before the event times out.

4.1 GPS Application

Our experiments use the Xemics GPS module [13], which is very power-efficient compared to other GPS units. However, its power consumption is still orders of magnitude more than the microcontroller. Furthermore, for a range of mechanical and environmental reasons, it is impossible to predict if the GPS will be able to acquire a lock and, if so, how long it will take. So, it is important to ensure that the GPS application operates as efficiently as possible.

We use this application to illustrate how applications interact with our middleware. Figure 3 shows how the GPS application activates the GPS and acquires a lock in the ZebraNet system. The process starts when the application places a GPS event in the timer queue by passing the function to the timer enqueue function (1). Our middleware continues normal operation until the system reaches the event time, at which point the event is moved into the function queue by the middleware (2).

The middleware executes functions in order from the function queue. Once the GPS event reaches the top of the queue, the middleware calls the designated GPS application function (3). This function has two primary tasks: turn on the GPS and schedule a timeout event. The timeout event is designed to shut the GPS off after two minutes so that we do not waste energy if there is little chance of getting a lock.

At this point, the information from the GPS is processed in interrupts and once our application determines that we have an accurate lock, it cancels the timeout event and places a cleanup event on the function queue (4). The cleanup event turns off the GPS, processes the data through the GPS application's custom compression algorithm, and if enough data has been acquired to run it through our middleware's generic, lossless compression algorithm (Section 4.3), it schedules a compression event (5). Finally, we go back to step 1 and schedule the next GPS event.

4.2 Network Application

Network services in our system are treated as an application. This application is implemented as a simple finite state machine, which handles peer discovery and data transfers between nodes. We use this application to illustrate how the application can function in different modes by utilizing the queue functions provided by the middleware.

The Network application uses the XTend^{TM} OEM RF Module [8] for its communications. This radio offers a long range and many advanced features including a powerful coprocessor to handle its software needs, but its power consumption for transmission at 1W RF power is more than 4.7W, which makes energy management of this module especially important. This peripheral is kept off most of the time by turning off its power supply.

The data to be transmitted is placed into a transmit buffer that is maintained by the application. The received data is collected by the interrupts routines and the buffer is available to the application only when a complete packet is received. The network service is scheduled at frequencies dependent on several variables we have selected to maintain a balance of data flow through the network.

This application acts both in blocking and pulling mode. It is a blocking application when it process radio packets preventing other application from delaying processing of the current packet. However between packets the network application exits and enqueue it self in the function queue by calling the enqueue function provided by the middleware. This allows other applications in the function queue to execute, which might further process, compress, or store the data.

4.3 Compression Application

To reduce the energy consumption of power-hungry transmissions and to improve throughput in an inherently unreliable network, we employ a novel variant of LZW data compression specifically tailored to sensor nodes [9]. Compression conserves a considerable amount of energy, and these energy savings are magnified as data is relayed from collar to collar through the network and as poor link quality requires the collars to retransmit packets frequently. We use this application to illustrate how a simple application can be implemented with the middleware.

Our system buffers GPS data and debugging information in non-volatile memory until it has accumulated two flash pages (512B) worth of data, and then it schedules the compression application by passing it to the function queue. The compression application will be executed in order according to the function queue.

5. MEASUREMENTS AND RESULTS

The core of the middleware includes the queues and the timer kernel. The application side includes applications that manage the communications, system monitoring, and peripheral applications. We have made preliminary measurements to evaluate the system's overhead and the memory requirements of these components.

5.1 Memory Footprint

With only 48KB of code space and 10KB of RAM, the MSP430F1611 has a relatively large amount of memory compared to other processors in the same class. However, it is still limited considering the multitude and diversity of the applications, and the frequent need for buffers in delay-tolerant networks. Middleware designed for delay-tolerant networks must have a small memory profile.

The left side of Figure 4 shows the code memory footprint of the system. The middleware code includes the func-

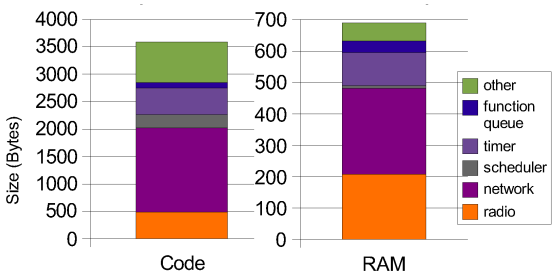

Figure 4: Memory footprint of major components of the middleware.

tion queue, the timer kernel, the scheduler, and the radio firmware. The entire system code consumes less than 2KB and the network application takes 1.5KB of code, leaving more than 44.5KB for other applications.

The graph on the right side of Figure 4 shows the data memory footprint of the system. The middleware and the network application combined use less than 700 bytes of RAM, of which the network application uses almost one third. This leaves more than 9300 bytes for other applications. The large memory needed for the radio firmware and the network layer are due to their packet buffers. This could later be changed to use a form of dynamic memory allocation to maximize memory usage.

5.2 System Overhead

Table 2: Overhead for queuing functions with fixed overheads.

Queue Functions	Overhead
Function Queue Enqueue	18 μs
Function Queue Dequeue	13.5 μs
Timer Queue Dequeue	21.5 μs

Our nodes have a 4MHz clock when the system is fully awake, and only a 4KHz clock when the system is in low-power idle mode. If the system overhead is large, problems could arise due to skipped interrupts, or delay in application executions. To avoid these problems, our middleware must have minimal delay to be non-intrusive to applications.

These measurements were made with an oscilloscope monitoring a user programmable hardware pin that is raised upon entry to the enqueue routine and lowered upon exit.

Table 2 shows that the function queue has a very low overhead for both enqueues and dequeues. As a comparison, the malloc function takes 49μs to allocate 56 bytes, and reading 56 bytes from the FLASH takes 3370 μs [7].

The overhead for the timer kernel to enqueue a function is variable due to sorting of the queue. The dynamic overhead for enqueuing is shown in Figure 5. Two scenarios are shown, one for appending a timed application to the end of the queue. In this scenario, the newly enqueued function is to be executed later than all the current scheduled functions. The overhead increases by 12us with each additional function already scheduled before.

In the same figure, the insert scenario shows delay for scheduling an application that should be executed before all the previous scheduled applications. Here, other functions needed to be shifted down before the new function can be scheduled. With each additional function in the queue that has to be shifted, enqueuing takes an additional 28μs. While this could be slow if too many functions are already

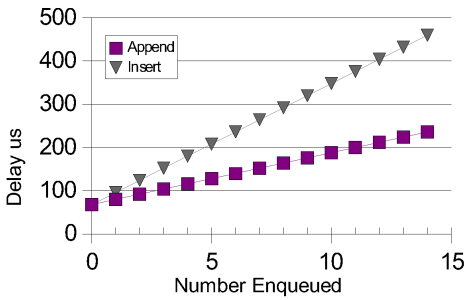

Figure 5: Timer kernel enqueue overhead for both inserting at the beginning of the timer queue and appending at the end of the queue.

scheduled, the function can finish within one time slot with more than 280 enqueued functions. This overhead is also less of an issue since functions are more likely executed by the scheduler after the application has already finished running.

5.3 Power Consumption

Table 3: Processor runtime power consumption. Measurements were taken by placing a current meter in series on the positive battery terminal.

Middleware State	Power
Application Running (Active Mode)	5600 μW
System Time Update (Idle Mode)	25 μW
Sleep (Idle Mode)	8 μW

In our implementation, the system spends most of its time in low-power idle mode. Our microcontroller has 4 low power modes and 2 possible crystal clocks. The middleware uses two of the low power modes and both clock modes for its operating modes.

The processor runtime power consumption is shown in Table 3. These measurements were taken by placing a current meter in series with the ZebraNet board's battery terminal. The battery was fully charged at 4.1V. In low-power idle mode, the system alternates between system time update and sleep. During this mode, the power consumption of the processor is scaled back to less than 0.5% of the power of the system in the active mode. Since the timer kernel is monitoring the application scheduling in low-power idle mode, the system maintains a low energy profile during the entire idle period.

6. RELATED WORK

As sensor networks have seen significant interest in recent years, much focus has been placed on middleware for them [1, 2, 3, 4, 5, 6]. However, much of the research has focused on close proximity, high-density networks. In contrast, our middleware system is aimed at long-running, low-duty-cycle, delay-tolerant sensor networks, where concurrent execution is less important. Below we compare and contrast some popular operating systems with the system presented in this paper.

TinyOS [4] is a popular and flexible operating system that has been ported to many platforms. It is similar to our system in that it provides the application with a simplified interface to the lower layers of the system. While a direct comparison of systems implemented in both systems is outside of the scope of this paper, the main differences stem from

intended applications. TinyOS is aimed at dense networks where many messages could pass through the system at the same time, resulting in its emphasis on event-driven processing. In many delay-tolerant networks, however, events are infrequent and rarely overlap. Thus, our system is more single-thread-based than TinyOS. One drawback of TinyOS is the possibility of missed events. A delay in TinyOS's event processing could lead to missed events, while our timer kernel processes delayed applications. Another difference is that TinyOS, while allowing reuse of the components during development, becomes a statically-linked application that is inflexible. Our system loads applications dynamically, allowing for dynamic scheduling and remote updates.

MantisOS [1] is similar to TinyOS but has greater emphasis on concurrency with dynamic user thread swapping. Our system executes a single application at a time, allowing only high priority services, i.e., system time updates, and hardware data receive, to interrupt. Our middleware has a very simple structure that schedules modular applications, using only 3500 bytes of code, less than one fourth the size of MantisOS.

Contiki [2] is another OS that is based on event driven cooperative multitasking. Our system differs in that it is focused more on long term scheduling than Contiki. Instead of middleware controls, our middleware employs the property of relatively long delay time in our target nodes to avoid hardware contention.

The Impala system [6] was used in the first two deployments of ZebraNet [14]. Our experiences with Impala led us to reduce the allowed concurrency in our current system to maintain simplicity. This simplicity pays off in both memory size and ease of development. In Impala, we took advantage of some of the hardware properties of a fixed number of data sources and sinks, by statically scheduling events. However, this was less flexible to changes. The current middleware presented in this paper goes further by taking full advantage of long-term, delay-tolerant systems and running a single-threaded system. This method offers more flexibility.

7. FUTURE WORK

The results from using our middleware system are encouraging, but there are many ways that we could improve on the current system.

One area of future work is priority queues. Currently there is no method to skip ahead in the queue to execute a function of higher priority besides canceling already enqueued entries. Another interesting area is a lean dynamic memory controller for the OS with possible extension into virtual memory. This would further simplify the applications' interface with different kinds of memory available to a node. For example, depending on the application's need, the memory could be allocated into the off-board data flash or on-board RAM. A third future work area is remote code updates, and our current system is designed with this in mind. Function pointers invoke all applications, so users simply need to update the pointer to invoke a new application. An application update consists of new applications being written to code space and scheduled by the scheduler when the update is complete.

8. CONCLUSION

This paper introduces a flexible middleware system for long-term, delay-tolerant networks. Our middleware simplifies software development for delay-tolerant networks, by only allowing one application to control the system at any time. This minimizes hardware conflicts between multiple applications. The unique system time allows an application to be scheduled well in advance, improving flexibility and reliability. The middleware's scheduler improves modularity of the applications, allowing for easy updates and improvements to the system.

This middleware system is also fast and thin, using little code and memory space. It provides a central location that can control the CPU sleep mode efficiently as well as monitor status to maintain sanity. Through its simplicity, it minimizes processing and monitoring requirements of the middleware and provides a platform to quickly produce reliable delay-tolerant networks. Overall, we believe that our system provides a flexible and simple platform to ease the development and improve the long-term reliability of long-running sensor node systems.

9. REFERENCES

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

[2] A. Dunkels, B. Grnvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Nov. 2004.

[3] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, 2005.

[4] J. Hill, R. Szewczyk, et al. System Architecture Directions for Networked Sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Apr. 2000.

[5] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, Oct. 2002.

[6] T. Liu and M. Martonosi. Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, June 2003.

[7] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet. In *Proceedings of the Second International Conference on Mobile Systems, Applications and Services*, 2004.

[8] Maxstream, Inc. XTend OEM RF Module: Product Manual v1.2.4. `http://www.maxstream.net/`, Oct. 2005.

[9] C. M. Sadler and M. Martonosi. Data Compression Algorithms for Energy-Constrained Devices in Delay Tolerant Networks. In *Proc. of the ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, Nov. 2006.

[10] Texas Instrument. Single-Cell Li-Ion and Li-Pol Battery Gas Gauge IC For Portable Applications data sheet. `http://www.ti.com/`, 2005.

[11] Texas Instruments. MSP430x16x Mixed Signal Microcontroller. `http://www.ti.com/`, 2002.

[12] T. van Dam and K. Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *The First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[13] Xemics. DP1201A, 433.92MHz Drop-in Module Product Brief. `http://www.xemics.com/`, Mar. 2004.

[14] P. Zhang, C. Sadler, S. Lyon, and M. Martonosi. Hardware Design Experiences in ZebraNet. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.