Measuring the Sap Flow of Eucalyptus Trees

Kaisen Lin
Neil Turner
Mark Kranz
Stephen Burgess

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-23
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-23.html

March 16, 2006

Measuring the Sap Flow of Eucalyptus Trees

Kaisen Lin, UC Berkeley
Neil Turner, UC Berkeley
Mark Kranz, University of Western Australia
Stephen Burgess, University of Western Australia
Philip Levis, Stanford University

Abstract

We describe the design and implementation of Savia, a sensor network system for measuring the sap flow of eucalyptus trees. Unlike prior sensor network systems, which generally take a vertically integrated approach and build everything from scratch, Savia is built out of several existing sensor systems. The difficulties we encountered when combining these systems demonstrate why a vertically integrated approach is more common. Based on our experiences composing Savia, we suggest two key abstractions that an OS should provide in order to allow sensor networks to grow beyond the current approach of writing everything from scratch.

1 Introduction

Wireless sensor networks can unobtrusively sample hard-to-reach places for long periods. Their decreasing cost and increasing sensing capabilities make them an attractive approach to a wide range of applications, including cane toad detection, microclimate measurement, phytoplankton migration, structural health monitoring, and condition-based maintenance. Experiences from these deployments have guided and motivated many research directions. For example, experiences from a sensor network in a redwood forest motivated the SNMS management system [26], while experiences from deploying heterogeneous sensor systems led to the Sympathy toolkit for diagnosing network failures [24]. Further examples include routing [28], platform power profiles [22], sensing modalities [4], programming models [10], and network architectures [9].

Conspicuously absent from this long list of lessons is any comment on the interfaces, structure, and abstractions of a sensor node OS. TinyOS, the dominant OS used in low-power sensornet nodes today, is completely event-driven [13]. There are several proposals for alternative approaches, including SOS [11], MOS [1], and OS* [16]. These proposals either address reprogramming efficiency for frequently reprogrammed systems (SOS) or argue for threaded rather than event-driven execution (MOS/OS*). Focusing on high-level (concurrency model) or low-level (linking model) issues, these research discussions have passed over what lies between, perhaps the most difficult aspect of an operating system: the abstractions it presents for hardware resources.

In this paper, we describe our experiences with *Savia*, a sensor network for monitoring the sap flow rates across a stand of eucalyptus trees in Australia. A Savia network takes synchronized sap flow readings across a network of sensors, logs these readings to non-volatile storage, and sends them unreliably to a collection point. Scientists can dynamically reprogram a Savia network, which allows them to request data from non-volatile storage, change the sampling rate, run diagnostics, adjust storage record sizes and what sensors to sample. We are in the process of deploying Savia in the wild: as of yet, it has only sampled the sap flow of trees in a greenhouse.

The current approach to building sensornet applications generally involves writing everything from scratch, leading to vertically integrated systems and very little sharing or comparative analysis. Looking at the wealth of readily available abstractions and implementations, we decided to try a different approach: we constructed *Savia* by building up from existing services and systems, including TinyOS and its MAC protocol, FTSP time synchronization [20] and the Maté virtual machine [18]. In one case — non-volatile storage — we tried three different existing subsystems before settling on one which met the application requirements. Savia runs on the Telos sensor platform.

Our experiences show why sensornet applications have been built as vertically integrated systems. One challenge we encountered was limited OS support for resource arbitration of underlying hardware: when two separate systems each independently think they can access the resource freely, one or both fail. In a vertically built system, application developers have precise control over hardware components, and so can build systems that cooperate effectively. In Savia, we encountered a multitude of problems that generally stemmed from interactions between all of the separate subsystems we collected together. For example, the storage system and network stack both access a shared hardware resource, the SPI bus, which they both try to use concurrently during startup. Getting both to operate in an application requires introducing additional, application-level logic to arbitrate this conflict, which happens to be a platform-specific artifact. Even then, improper use of chip select pins led to bogus data bytes that the non-volatile storage layer must discard.

Although relatively unnecessary in the traditional computing realm, efficient power management is critical to the lifetime of a sensor network as well. Because of the precise control of hardware components available to vertically built systems, application developers are able to more fully understand the ramifications of putting subsystems to sleep in such a way that will not interfere with the functionality of the system. With systems built out of existing services, careful consideration must be given to putting components into sleep mode. For example, turning off the radio for extended periods of time will interfere with any time synchronization services that send periodic time packets.

We do not consider these difficulties to be faults against the designers and implementers of the subsystems we used. Instead, these bugs and problems were all the result of basic limitations in the OS abstractions. For example, TinyOS's event-driven execution model does not have good mechanisms for managing resource contention, which is problematic when two subsystems, such as the radio and non-volatile storage, both require the same resource. These complexities and interactions suggest requirements that future sensor network operating systems may need to consider.

This paper has two research contributions. First, it describes Savia, a flexible sensor network application designed to support biological experiments, as well as includes a precise set of requirements for these classes of networks. Second, this paper notes the difficulties encounted when implementing Savia and suggests three things that future sensor network operating systems should consider providing as basic abstractions.

In the next section, we provide background information on TinyOS and the other existing systems that Savia builds on. In Section 3, we present the scientific methodology of sap flow experiments and the requirements this methodology places on a sensor network system, including synchronized sampling, sensor configuration, and reliable data collection. In Section 4 we present the design of Savia, a TinyOS application that meets these requirements. In Section 5, we evaluate Savia in terms of its requirements and present initial scientific data. We present related work in Section 6, discuss what our results and experiences in Section 7, and conclude in Section 8.

2 Background

In this section, we describe the systems underlying Savia, including the Telos revB platform, the TinyOS operating system, the Maté virtual machine architecture, the monibus sap flow sensor and the flooding time synchronization protocol (FTSP).

2.1 Telos revB

The Telos revB platform is designed for sensor systems research. Its distinguishing characteristics include a very low power sleep state, 10 kB of RAM, 48kB of program memory, 1MB of non-volatile storage, a USB connector, two sensor expansion ports, and an 802.15.4 radio [22]. Telos has the most RAM of current platforms: the next closest is the micaZ from the Crossbow corporation [3], which uses a different microcontroller that has 4kB of RAM but 128 kB of program memory.

Telos uses an MSP430 F1611 microcontroller, which has two peripheral buses, USART0 and USART1. Each bus can operate in three bus modes: UART, SPI, or I^2C. The mode can be changed at runtime with configuration registers, but as these protocols share physical pins, each bus can only operate in one mode at any time. The Telos' USART1 bus is dedicated to the USB connector. The USART0 bus is shared for all of the other peripherals, including the radio, non-volatile storage, and sensors. Both the radio and storage are SPI devices, while external sensors vary.

2.2 TinyOS

TinyOS is an event-driven operating system designed for low-power sensor devices that have limited resources [13]. It is written in nesC, a C-based component languages. Because microcontrollers generally do not have memory protection, TinyOS is not an OS in the traditional sense. Instead, TinyOS is a collection of software components that application components can use. The nesC compiler only includes the OS components that an application actually uses: unneeded services do not consume RAM or program memory [8].

One way that TinyOS conserves memory is by having a completely event-driven execution model. System calls, such as sending a packet, are split-phase. The request to start the operation returns immediately, and the system issues a callback when it completes. This approach means that TinyOS has a single thread of control and therefore a single execution stack, rather than allocating multiple stacks for multiple threads.

Because operations do not block, contention for a resource is generally handled in a first-come-first-served policy. If a device is busy, then it returns an error code on additional requests. For example, if the radio is sending a packet and a component calls the send command, the call returns FAIL. TinyOS expects application code to institute arbitration policies when needed.

2.3 Maté

Maté is an architecture for building application-specific virtual machines (ASVMs) [18], which are bytecode interpreters. The architecture has two parts. The first is the

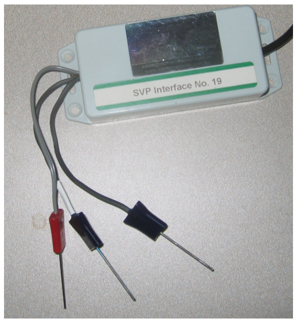


Figure 1: The Monibus sap flow sensor. The heat pulse comes through the needles and the device begins calibration. After calibration, the needles can be queried for a velocity value.

build system, which generates an ASVM from a high-level description of its needed abstractions. The second part of the architecture is the programming toolchain. The toolchain compiles a user script into the instruction set of the ASVM and installs it in the network.

Every ASVM includes a set of a core services, such as concurrency management, type checking and code propagation. Based on the description of needed abstractions, the build system adds a set of *extensions* to the VM, including the instructions the desired language compiles to, code handlers for the events an appliction needs to execute in response to, and functions that provide application-level abstractions. An ASVM is a TinyOS binary.

2.4 Monibus

The Monibus sap flow sensor from Monitor Sensors is commonly used in automated sap flow measurement by biologists, government, and industry (timber). The sensor has an external power supply and solar recharger. It measures sap flop using a three pronged device shown in Figure 1. One prong is a heating element, and the other two are temperature sensors. Installing the sensor involves embedding all three prongs in a tree, with the two temperature prongs being above and below the heating element. To take a measurement, the sensor emits a heat pulse, then measures the time it took the pulse to reach the two sensors. It then processes the time stamps with respect to eucalyptus specific properties to produce a meaningful value. All in all, sampling the sensor takes approximately two minutes. The Monibus sensor has a UART-like interface. Intended for interactive use, it receives commands and outputs data in ASCII format. This requires connecting it to a Telos expansion port and configuring USART0 to be a UART.

2.5 FTSP

The flooding time synchronization protocol (FTSP) synchronizes all nodes in a multihop network to a single reference point (the root node) [20]. The protocol works by taking high-precision timestamps at both the receiver and sender at the first bit of the physical-layer signal synchronization sequence. The transmitter embeds this timestamp in the footer of its packet, and the receiver compares the transmitter's time stamp to its own.

FTSP adjusts for clock drift between nodes by computing the drift with a linear regression over the past 8 data points. The limited precision of this calculation means that even if nodes are well synchonized, they will drift if they lose connectivity, but this drift is much smaller than the drift of their clocks (on the mica2 platform, at most 40μs per hour, rather than 40μs per second). Experimental results on the mica2 platform (which uses a different radio and microcontroller than Telos) show these techniques allow a large, multihop network of 60 nodes to all synchronize within 10μs.

3 Requirements

In forest ecosystems, approximately 30% of total rainfall returns to the atmosphere by plant transpiration. For areas prone to drought or areas with increasing water demands due to population growth or agriculture, understanding what affects this large input to available water can greatly affecting managing water resources. Transpiration is, for the most part, a result of tree sap moving water from root systems to leaves.

Being able to accurately measure sap flow rates can therefore allow ecologists and tree physiologists answer key research questions that can guide water management and land development decisions. Does plant height, and therefore age and species matter? Also, ecological questions related to forest hydrology focus on resource partitioning among different species combinations. For example, is niche-filling by diverse plant assemblages the key to sustainable resource capture by native vegetation?

There are three common experimental methodologies for monitoring plant transpiration, which operate at very different levels of granularity. Broad-scale measurements that measure changes in soil moisture can obtain data on entire stands of trees but cannot distinguish the role of different species (e.g. [5]). At the other end of the spectrum, infra-red gas analysis of leaf-level gas exchange can give precise data on a small area but cannot scale up to even cover a small branch. A third approach is to measure sap flow which is easily scalable from individual branches to whole stem and then to multiple trees [12].

Sap flow measurements also provide the advantage of working at both the branch and stem level because it permits partitioning of water fluxes in different canopy strata and reveals within-tree source-sink and water storage dynamics. Plant biologists have put in huge efforts to make sure their sensing technology is reliable, robust, and properly calibrated.

Plant biologists have put in huge efforts to make sure their sensing technology is reliable, robust, and properly calibrated. But a recent review of canopy research [6] shows that one of the greatest challenges in monitoring a forest environment is the extensive cabling to control and collect data. A recent experiment in a Californian redwood forest, for example, required what amounted to 7km of wire to study 9 large trees [2]. Such cabling is costly, causes signal degradation, is prone to damage by animals, weather (e.g. falling tree branches) and vandals, and poses incredible routing challenges for work in swaying canopies. The above constraints introduce error in the data because only small clusters of sensors can be deployed from which data are extrapolated to cover larger areas. Moreover, sensor placement is not determined by 'randomized design' or 'representative sampling' but 'where the cable will reach'.

3.1 Requirements

The Savia project is a close and ongoing collaboration between researchers in plant biology and experimental wireless sensor technology. In order for Savia to be useful in instrumenting individual trees and forests with sap flow sensors the plant biologists provided these high-level requirements:

- **Existing Sensor Hardware.** Must use sensors already tested, used, and accepted by the community. For this deployment, it is a Monibus Sap Flow sensor.
- **Synchronized Sampling.** Sensors must be synchronized when sampling and done on an interval between 5 to 30 minutes for several months at a time.
- **Network Retasking.** Ability to easily change the experiment, i.e. reprogram the sensors with clear semanticsafter they have been deployed (without having to touch the sensors, which may be in hardly accessible places). This includes adjusting the records, recovering data from flash, and changing the sampling intervals.
- **Non-volatile Storage.** Enough local storage for months-worth of data. There must also be flexible record sizes. Ultimately, this system will be general to different environments and different sensing needs. E.g. meteorlogical sensors are an obvious next step.
- **Completely Wireless.** No power, data, or control cables from base station to sensor locations nor between sensor locations.

The 5-minute interval between samples comes from the heat-pulse method for measuring sap flow as well as the frequency of independant events produced by trees and their ecosystems. Experience suggests that more than 5 minutes is required for the heat to dissapate in woody plants between heat pulses.

Synchronization of measurements at different physical locations plays an important role in deployments because of the need to correlate measurements. For example, if one hypothesizes that incident light is what triggers transpiration, then one would like to be able to look at light measurements, sap flow measurements, and perhaps humidity and temperature measurements all taken at the same time. This variety of sensor types again speaks directly to the desire to expand the system to include all sorts of meteorlogical sensors.

The overarching requirement is to provide at least the same performance and functionality that wired solutions currently provide to plant biologists while introducing a whole new set of possibilities (e.g. sampling density and ease of reprogramming) through the wireless technology.

4 Design

Measuring sap flow is an example of a low-rate habitat monitoring application. Therefore, we chose to use the Telos platform [22] as the basis for Savia, as habitat monitoring applications [25] guided its design. As existing reprogramming systems, such as Maté [18], TinyDB [19], and SOS [11] commonly cite RAM as the major limiting resource, we believed that Telos' comparatively large RAM (10kB) would be important. Additionally, Telos' 1MB of external flash is larger than that of other platforms, allowing experiments to run longer, and its 802.15.4 radio has reasonable range. In combination, these factors made Telos appear to be the best platform for our application.

We chose TinyOS as an operating system because it has the best Telos support, including tree-based routing. The dynamic environment within a eucalyptus forest requires network retasking to adapt to new situations. Because of this, we decided to use Maté [17] as our application level programming system. Maté allows users to reprogram the network using TinyScript, a BASIC dialect; as one of the biolgists who will use the system already knows BASIC, this seemed like a good fit.

Maté comes with many sample extensions, such as timers, basic sensors, and routing. Savia, however, requires additional extensions that stem from application requirements. After discussions with the biologists on what programming interface would be easiest, we settled on three abstractions, shown in Figure 2 that would constitute the ASVM extensions:

Synchronization: users can write a code routine that runs in response to a synchronized timer. The nodes maintain a globally synchronized timebase, which they each locally use to decide when to fire the timer. All of

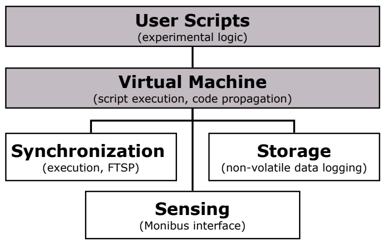


Figure 2: The Savia software architecture. Scientists program the Savia network in high-level scripts, which compile to an application-specific virtual machine. Supporting redwood sap flow experiments as an application domain requires three VM extensions: synchronization, storage, and sensing.

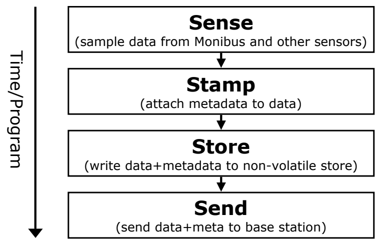

Time/Program
Sense
(sample data from Monibus and other sensors)
Stamp
(attach metadata to data)
Store
(write data+metadata to non-volatile store)
Send
(send data+meta to base station)

Figure 3: The four basic steps of a sap flow experiment. The sense, stamp, and store steps use Savia's VM extensions: send is the basic TinyOS tree routing algorithm (MultiHopLQI). The third Savia extension, synchonrization, controls *when* the program runs.

the nodes run the code routine at the same time.

Storage: users can log readings to non-volatile storage, from which they can be retrieved later. Logging in this fashion means that if some readings were lost due to networking problems, the user can easily recover them by writing a simple script. The storage system automatically includes important metadata, such as timestamps and sequence numbers.

Sensing: the sap flow sensors that biologists use follow an ASCII-based protocol intended for interactive human use. Experiments require that biologists be able to adjust the time intervals between starting a heat pulse and reading the result, so user scripts must be able to control this as well.

In deployment, the Savia nodes form an ad-hoc collection tree which is rooted at a base station. The base station is a Sharp Zaurus PDA running Java and Linux. The base station receives data from the network and periodically uploads the data to a server over a cellular link.

4.1 Application Scripting Interface

Figure 3 shows the basic program structure for a sap flow experiment. The first step is *sensing*, which samples the monibus sap flow sensor. *Stamping* is the second step, where the reading is given metadata such as a timestamp and sequence number. The third step is *storing*, when the program logs the data to non-volatile storage so it can be retrieved later. The final step is *sending*, which forwards the data up an ad-hoc collection tree. Each of the three Savia extensions has an associated scripting interface. After lengthy discussions with biologists for this deployment, we settled on the system calls shown in Table 1.

Savia provides a simple record abstraction to flash memory. Records consist of metadata and one or more 16-bit data values. The fformat() call allows a script to format the flash for a certain record size. Making this a VM function, rather than a compile-time constant, allows biologists to reuse a deployed network for multiple experiments. Having a fixed size allows scripts to be able to efficiently read records with the fread() call, as all Savia has to do is calculate an offset, rather than maintain indexing structures. The fwrite() call writes a record to storage. All writes are sequential. The stamp() call takes a buffer filled with sensor data and prepends metadata to it.

The interface to the synchronized timer has two calls, start and stop. The start call allows a fidelity of minutes and seconds. From an scientific standpoint, seconds are not very useful: the sensors take several minutes to generate a reading. However, it turns out to be important when first deploying the network and testing that it works correctly.

The Monibus device interface includes basic control and collect functions. The first step to taking a sample is emitting a heat pulse with the hrmpulse() function. After waiting for the pulse to travel (a few minutes) so the sensor can calculate sap flow, the script can then read the sap flow rates. The interface also allows a user to sample the voltage of the Monibus power source, which lets a biologist easily determine when the power supply needs replacing.

4.2 Non-Volatile Data Storage

Of the extensions, the storage system had the most iterations. We started by using the Matchbox File System [7] to store results of sensor readings, but found this to be overly complex. The Matchbox interface does not provide easy random access, which is needed for scripts to be able to read data. Therefore, storing all of the data in a single file was not feasible. On the other hand, Matchbox also requires file system metadata blocks, which imposea a large overhead on storing many small files. The Matchbox file abstraction, while useful for some application domains, was not well suited to the needs of long-term, low-rate monitoring and data collection. The biologists preferred access to specific data, not specific files.

Flash Storage	
fformat(int)	Formats the flash given a record size.
fwrite(buffer)	Writes a buffer to flash as a new record.
fread(buffer, int)	Reads a record into a buffer given a record ID.
stamp(buffer)	Takes a buffer and prepends it with some metadata.
Synchronized Timer	
startsynch(int, int)	Starts the synchronized timer with parameters minutes and seconds.
stopsynch()	Stops the synchronized timer.
Sap Flow Sensors	
hrmpulse()	Fire the heat pulse.
hrminner()	Get the value of inner sap flow velocity.
hrmouter()	Get the value of outer sap flow velocity.
hrmvolt()	Query the Monibus device for its supply voltage.

Table 1: Savia programming interface.

Timestamp	
Sequence Number	
Data_0	Data_1
...	Data_{n-1}
Voltage	Status

Figure 4: The Savia record layout. Each row is four bytes: the timestamp and sequence number are both 32 bits, while the voltage reading is 16 bits. The number of data entries depends on the formatted data size.

We considered writing a Savia-specific interface on top of the flash chip, but this ran contrary to our goal of composing Savia out of existing components and systems whenever possible. We therefore settled on borrowing the block abstraction underlying the Deluge reprogramming system [14]. Savia provides a logging system on top of the block interface, which has two parts:

- **Record structure** - formats sensor data into a record.
- **Sequential storage** - writes records to end of log.

4.2.1 Record Structure

The first component of the storage system takes a set of sensor readings and formats them into a storage record. Figure 4 shows the record layout. Scripts put data into a Maté buffer (a vector), and call the stamp() function to format the buffer contents into a record. The stamp function prepends buffers with useful metadata such as timestamps, sequence numbers, and health information to create a complete record. Timestamps are useful for determining when exactly the sensor value was taken in case sampling was done slightly out of sync. Sequence numbers are used to see if any records were missed or misaligned. Finally, the health information is used to report the current status of the system so appropriate action can be taken in case of a malfunction. Because status information can be application specific, it can also be used to represent other measurements. In our current system, some of the status information records the voltage of the mote platform.

We originally had through that metadata stamping should be automatic, so the script writer did not have to be aware of it. Unfortunately, this creates a few problems. One issue was that the both the store and send operations need to have the same stamping metadata, so they can be correlated. Stamping them separately introduces delay, which can create inconsistencies in timestamps. This then raises the question of when to stamp the metadata, which becomes trickier if scripts do not follow the basic structure show in Figure 3. For example, scripts may read data out of storage then send it; in this case, clearly the send operation should not stamp the data. Making the stamp operation a separate function also separates data and metadata, leaving the send, read, and write implementations independent of particular data formats.

4.2.2 Sequential Storage

The second component to the hardware level is the raw flash logger, which underlies the fread(), fwrite(), and fformat() functions. It simply takes a pointer to an array of bytes and appends it to the last written position. It is slightly different from a traditional data logger in that it expects fixed size records. The flash logging component maintains a small amount of non-volatile state so it can recover properly from losing power. This metadata is a handful of bytes, stored in the small on-chip flash. The metadata includes a format bit, the record size, the current sequence number, and current write offset in external flash. The format bit is only cleared when the mote is physically reprogrammed, and it is set when the mote boots up for the first time, preventing a script from accidently reformatting the flash if the mote is reset.

4.3 Synchronized Sampling

The synchronization component is based on top of the Flooding Time Synchronization Protocol (FTSP) [20].

We chose FTSP because it is robust to node failures and scales well to a large number of nodes. It is also extremely accurate and adjusts for clock skews that are inherent in the 32kHz clocks of the Telos platform. FTSP was originally designed for the mica2 platform; we obtained a prototype version for Telos from the authors at Vanderbilt.

FTSP provides an interface to sample the global time of the network based on the time of the root node. To provide a synchronized timer, Savia uses a local timer that periodically checks the global FTSP time. If the interval is large and Savia is far away from the next deadline, then by default Savia sets the local timer to fire at a low rate. The default rate is set at compile time, and the standard value is twenty seconds.

If the local timer fires and Savia is close to the deadline, then it sets the local timer to fire slightly before (20ms) the timer deadline. This padding allows the underlying logic to avoid wrap-arounds and other edge cases caused by jitter or latency that can introduce significant complexities. If the local timer fires and Savia is very close to the deadline (within the padding period), then it fires the synchronized timer. Rather than a periodic local timer, Savia always uses one-shot timers. As the interval for the local timer is determined completely by when the next percieved synchronized timer must fire, this works well while avoiding issues caused by assuming the periodicity of local clock corresponds to the global clock.

Choosing the right default rate is a tradeoff between accuracy and cost. If the default rate is too small, then we waste energy by waking up prematurely multiple times. The benefit of this though is it can adjust to sudden changes in the global time. The more sampling that is done, the more accurate the synchronized timer will be. On the other hand, if the default rate is too large, then we always fire the timer slightly before the intended deadline. This means if we read the global time incorrectly, we will be completely off the mark with no hope of adjustment. Thus the tradeoff here is energy versus accuracy.

The other challenge in implementing a synchronized timer was making it resilient to temporary loss of synchronization as well as rapid shifts in synchronized time. More specifically, outlier global time readings should not confuse the synchronized timer. To accommodate this, the synchronized timer code keeps state about its previous readings. This information is not used to decide when to next fire a timer, rather, to validate whether the global clock is accurate. When Savia samples the global time, it checks that it is greater than the previous time, but not greater than some epsilon ϵ, which represents an upper bound. If the global time reading is not within the range $(last, last + \epsilon)$, we classify it as invalid. However, in case there has been a significant shift in the synchronized time, we record the invalid time in case the next reading agrees with it. In the case we happen to read two strange readings in a row, Savia assumes there has been a genuine time reference change and it should adjust accordingly.

In addition, because the global network time is stored as a 32-bit number, Savia must keep the synchronized timer consistent when the global time wraps around by considering the time passed after the last firing and before the wrap-around (2^{32} modulo the synchronized timer period). To handle this corner case, whenever Savia detects a time that seems to go backwards in time, it checks if it is, considering wrap-around, within the range $(last, last + \epsilon)$. If so, it incorporates an offset of (2^{32} modulo period) into its global time. Merely checking if time has decreased is insufficient, due the occasional presence of faulty readings.

4.4 Sap Flow

Another requirement of this system was the use of an external sensing device. We attached our mote to a heat pulse sensor that speaks the Monibus protocol. The values returned from the Monibus are represented as ASCII strings. Our design of integrating the sap flow sensor is comprised of two elements:

- **Bus Arbitration** - Interfacing the sap flow sensors with the hardware.
- **Monibus Protocol** - Communicating with the sap flow sensors.

4.4.1 Bus Arbitration

In order to even begin using the sap flow sensor, we had to implement our own data bus arbitration component because it shared physical pins on the Telos MSP430 microcontroller. The layout is shown in Figure 5. MSP430 has a set of pins that can be treated as either an SPI bus, I2C bus, or UART bus. Each bus type has its own set of communication protocols. For example, SPI uses a 4-pin connection (data in, data out, clock, chip select), where as UART only uses a 1-pin connection (data). Because of resource sharing, we wrote a special component specifically to handle changing the bus mode, using the pins, and then resetting the bus mode when using the Monibus device. This is important because the CC2420 and the flash chip both used SPI, where as the Monibus device used UART.

4.4.2 Monibus Protocol

The first step to interfacing the Monibus device with the Telos was building a component to use the protocol properly. The Monibus device accepts several 1- to 3-byte ASCII character commands. Thus to send commands, we simply built a component to send ASCII characters to

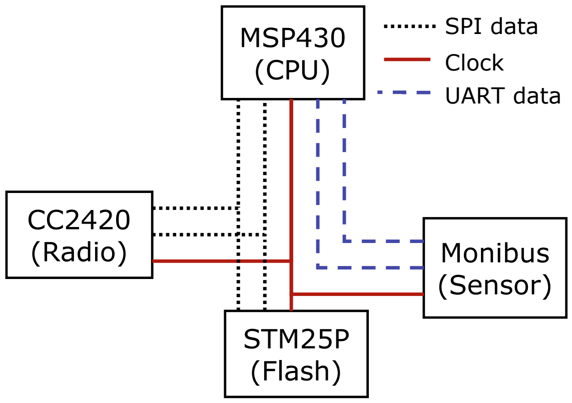


Figure 5: Serial interface of the MSP430. The MSP430 has one set of shared pins that connect with the radio, flash, and Monibus sensor. The radio and flash both use SPI, while the Monibus uses UART. The Monibus shared the clock line, but does not use it.

the device and then wait for a response. Receiving data from the Monibus in an efficient manner was slightly more complicated. Because the Monibus communicates with ASCII instead of binary data, storing it in its raw ASCII form is very inefficient. In addition, the ASCII characters represent floating point numbers.

We resolved the size constraint by first converting each byte upon receipt into an integer type. Then once all the bytes are received and converted, the values are combined into one integer value. The floating point problem is resolved by ignoring the decimal point. The precision of each measurement is uniform and known so the decimal place can be reinserted when the data is processed. We did not save any unit information because the script writer is expected to know the units based on the record layout that they specify themselves.

4.5 Integration

The integration of all these systems also influenced our design because of the interaction complexities. An issue we ran into was the initialization of hardware components that used the same data bus. This anomaly sometimes causes only a parial set of components to be initialized. The problem occurs when two components race for the data bus, and one of them fails to initialize properly. Because there is no retry mechanism on initialization, some components simply fail to work. We searched for documentation regarding this issue, but none were found. After troubleshooting for a week, we finally discovered the problem when one of the authors of the CC2420 radio code happened to mention that there was this problem. Thus we were finally able to correct this by serializing the initialization sequence. For example, we had the external flash initialize only after the radio has finished. The mechanism we used was the TinyOS SplitControl interface.

Another issue that influenced our design was the shared nature of the SPI. Because we had to be careful of when we used certain components, we implemented several script commands to explicitly enable or disable the monibus and radio. This was necessary because there was an issue regarding the SPI bus not being flushed after using devices on it. Thus subsequent readings from flash would read garbage from the previous Monibus usage. To correct this, we simply reset the SPI bus before using the flash, thus flushing it out before reading data.

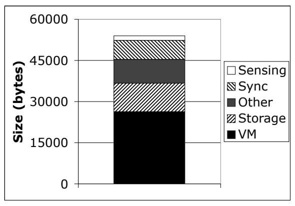
As a way to detect errors early in the deployment, we also instrumented several error bits in the status piece of the record. Errors included a problem with the flash, errors with the monibus device, or even errors with the synchronization. This means even in the event of no errors, each record will still have an overhead of 2 bytes. However, this was a worthy tradeoff because it allowed us to pinpont exactly when the error occurred and take proper action.

Our requirements also state that our system needs to be running for months at a time. Because of this, we had to explicitly sleep all components that were not in use, including the radio. Completely disabling the radio is a challenging task, since there are many underlying services that rely on radio to function. For example, long periods of radio silence will decrease the effectiveness of the synchronization and routing algorithms, as they rely on the regular exchange of radio messages. There are also expected software side-effects, as the radio driver responds differently when the radio is powered down. We found that ensuring the radio is active for some time before using it was an effective way to solve the first problem - giving radio-reliant services enough time to refresh stale network state information. To overcome the latter problem, we needed to make the radio switching software aware of the affected services, and restart them if they crash or hang.

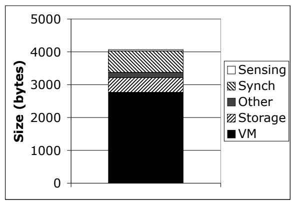
5 Evaluation

We evaluated the system based on how robust the system was and how well it met the requirements. Sensor network deployments in deep environmental settings often times have less than ideal reliability. Thus most of our evaluations ran for long periods of time without human intervention. Before evaluating each component in detail, we first examine the varying sizes of each component as shown in Figure 6. Because of the limited ROM size of the Telos platform, keeping components as small as possible is important. We also saved some space by cutting out unnecessary opcodes in the TinyScript language like the exponentiation function, which carries with it the weight of a full mathematical library.

From the data, we can infer from the ROM size that



(a) Code Memory



(b) RAM

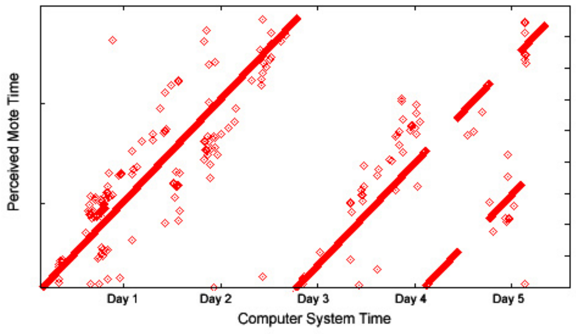
Figure 6: Code and RAM space utilization of Savia's components. VM is the core Maté VM. Storage is the entire storage stack (page interface, record structure, and sequential logging). Synch refers to the time synchronization stack as well as its Maté code routine. Sensor is the Monibus sensor drivers, and Other represents other, micellaneous parts of the system, such as on-board ADC sensors.

much of the complexity comes from the flash component. This is because we decomposed it into three layers to encourage reusability. Based on RAM size, we consume a lot of RAM in the synchronized timer. This is attributed to the fact that we are keeping around a lot of timer state, including some invalid readings. Because this data is to keep the global network time robust against network failure, this cost is unavoidable.

5.1 Synchronized Sampling

We evaluated the time synchronization component on two different levels to measure its effectiveness. First, we evaluated the time synchronization component by itself. We instrumented the code to send a UART packet everytime our local timer fired. The packet contained the perceived global time, the remaining time left, and a bit as to whether or not the mote thought it was synchronized. On top of this, we created a synchronized timer handler, which simply sampled its voltage, wrote it to external flash, and then sent the data to the UART. The computer connected to the UART would receive the message and timestamp when the message was received. We then deployed it on a 28 mote Telos testbed for several days. Our initial unfilitered results shown at the top of Figure 7 were surprising. There were many spurious global time readings, and although not shown on the graph, there were also many times the mote could not synchronize. The disconnectedness in the graph is caused by the root node temporarily losing connectivity. In this event, the next node becomes the time base. When the original root node recovers, then all other nodes return to synchronizing with the original root node.

However, after we began to filter out readings, we had much better results as shown at the bottom of Figure 7. The spurious readings that were above the expected time were recorded because they were within our threshold. This could have been filtered out with a tighter threshold. The spurious readings below the expected time represented a corner case in our filtering. In that situation, there were two invalid readings that were valid with respect to each other. Thus we assumed it to be a valid time shift. However, this is relatively rare, and this situation is quickly corrected. It also did not adversely affect our actual synchronized timer.



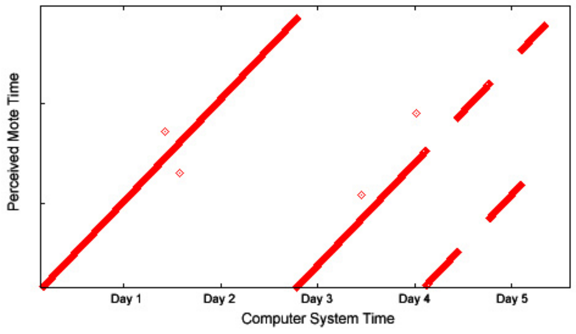


Figure 7: The top graph shows time synchronization of one mote without our filtering of invalid readings. The bottom graph shows the results after we filtered out invalid readings.

After evaluating details specific to time synchronization, we evaluated the effectiveness and accuracy of our synchronized timer context as a whole. Our results are shown in Figure 8. There was a small amount of jitter in some areas, but the system was able to resynchronize despite this problem. The problem was caused from the root node shifting out of phase. However, we can see that all the other nodes adjusted properly. With the help of sequence numbers, this minor jitter was not very problematic because the data could still be aligned to the correct set. Thus we were able to offer a fair amount of synchronization at a fine level of granularity.

5.2 Network Retasking

Our final goal was a simple script to take sensor readings and route them to a base station. Figure 9 shows the script we ran to exercise all parts of the system. Access to the UART, radio, radio control, flash, and timers were all available at the script level through Savia giving our system a large selection of network retasking capabilities.

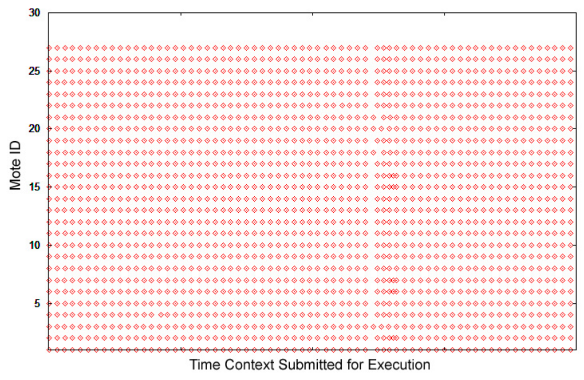


Figure 8: A snapshot of when each mote submitted its context for execution. Each point represents when the UART packet was received for a specific Mote ID. The sampling interval for this experiment was 5 minutes.

Other scripts that have been used in actual deployment are more or less complex, depending on the experimental needs.

5.3 Storage Availability

We evaluated the storage system on its durability. We ran a basic sampling script for six days on a pair of Telos motes running on AA batteries. During the experiment, we took out the batteries for a few minutes multiple times for each mote. At the end of the six days, we used the scripter to query out random samples towards the beginning, middle, and end via the script shown in Figure 10. By comparing the steadily decreasing voltage values, we were able to successfully verify that all our data was there even though the mote had been interrupted several times. It is important to note that the Maté code dissemination protocol was robust enough to allow motes that exit and re-enter the network to automatically recover and run the latest code.

We also verified that the error reporting mechanism was working. We intentionally made the page sizes small so that we would exhaust our supply of flash memory. By scanning the packets being collected and examining the status bits, we were able to determine that the flash memory on the mote indeed had been exhausted.

With record sizes on the order of 20 bytes and 1MB of external flash, we can store approximately 60000 records without running out of space. If we sample every 5 minutes then we can keep the system running for over 200 days. This is more than enough to satisfy the requirements of the deployment.

5.4 Integration

To evaluate our system under a full deployment scenario, we deployed 3 Telos motes inside a potted plant at a university green house using the full sensor suite. Our sensor suite included temperature, light, humidity, and sap

```
buffer msg;
private SAMPLE;
shared seqNum;

! Message ID
SAMPLE = 2;
! Clear the buffer
bclear(msg);

! Activate monibus
monibuson();
! Begin pulse
hrmpulse();

! turn off radio during this time
radiosleep(100);
! Wait for pulse (105sec)
wait(1050);

! Message header
seqNum = seqNum + 1;
msg[0] = SAMPLE;
msg[1] = id();
getnetworktime(msg);
msg[4] = seqNum;

! samples
msg[5] = int(hrmvoltage());
msg[6] = int(temp());
msg[7] = int(humidity());
msg[8] = int(photoactive());
msg[9] = int(totalsolar());
msg[10]= int(hrminnervelocity());
msg[11]= int(hrmoutervelocity());

! Send samples over radio and uart
broadcast(msg);
uart(msg);

! Write the samples to
!   persistent storage
fwrite(msg);

! Wait a short period to ensure
!   all samples get through
!   then shut down
wait(150);
monibusoff();
radiosleep(478);
```

Figure 9: Complex TinyScript code used in actual deployment. It simultaneously exercises many facets of the Savia system.

```
buffer b;
for i = 0 to 5000
  fread(b, i);
  send(b);
next i
```

Figure 10: TinyScript code to recover data from flash.

velocity readings. By using as many components as possible, we could evaluate how effective our resource arbitration was. This system was deployed for a 60 hour period. During this period we verified that the nodes remained in sync, that the power management was effective at preserving battery life, and that nodes that were forcefully rebooted could automatically rejoin the network.

We also integrated several gateways, including the Zaurus and a traditional PC setup. While these are currently running with one or two node networks, given the previous tests, we are confident that they can be easily expanded. Future deployments will consist of a larger number of nodes, professionally installed and calibrated, with a web-service to view the data through the gateway.

External sensor devices can also be a source of significant power consumption. Although our sap flow sensor returns to a low powered 'sleep' mode, this may not be true for all monibus based devices. Power consumption is important because the more power that is drawn the shorter the lifetime of the deployment. Therefore we examined how much energy was being drawn and what components were drawing it. As shown in Table 2, the Monibus heat pulse draws a significant amount of power when active, and thus should be put to sleep whenever possible. Even with solar rechargers, the results show we are near the edge of our power budget. Thus an unlucky string of cloudy days could adversely affect the sap flow sensor.

	Time (%)	**current) (mA)**	**power (mW)**
Mote (-3.3V)			
Radio active	30	20	-3.3
Radio inactive	568	1	-3.124
Sensor bundle	1	0.5	-0.003
Flash storage	1	20	-0.11
Total			-6.54
Monibus (-12.0V)			
Heat pulse	2.5	667	-33.35
Measurement	105	5.5	-11.55
Inactive	492.5	0.2	-1.97
Total			-46.87
Solar (+12.0V)			
Full Sun	7200	60	+60
No Sun	79200	0	0
Total			+60

Table 2: Power budget for a typical deployment. Samples are taken once every 10 minutes, and there is at least two hours of partial-solar coverage per 24 hours.

6 Related Work

Sensor network deployments related to environmental monitoring have been done in the past. The most recent being a redwood tree deployment [27] measuring microclimates along redwood trees in Sonoma, California using Berkeley Mica2Dot Motes manufactured by Crossbow. In that deployment, traditional climate variables such as temperature, humidity, and light levels wre measured. However, in our deployment, we used sap flow sensors which not only measured variables outside the tree, but also variables inside, making it a much more intrusive deployment requiring in-situ adjustments.

There have also been similar wildlife tracking deployments such as ZebraNet [15] and Great Duck Island [25]. In the ZebraNet deployment, sensors, including the base station, were actually attached to the zebras. Their deployment, however, used very customized hardware and a vertically built system, specific to their application needs. In our deployment, we took off-the-shelf devices and services, and combined them into a working system. Because of this, we were not guaranteed that the devices would work properly with each other. In the Great Duck Island deployment, scientists were interested in the occupancy of nesting burrows. The deployment had to be done carefully to not disturb the nesting petrels.

Similar work has also been done in network retasking such as VM* [16], and Deluge [14]. VM* is framework for programming sensor nodes that includes a Java Virtual Machine (JVM), code synthesis tools, and an energy efficient incremental linker for retasking. VM* programs are written in the Java language. Because VM* programs represent only the necessary components for a specific deployment, it has very compact code size. However, because it runs a JVM, all programs must be written in the Java language. By using Maté we were able to take advantage of a very simple scripting language with just as much expressiveness. Deluge is a binary image reprogramming tool. This implies that any retasking must be done at the nesC level. Due to nesC's event-driven programming model, it is not well suited for biologists who simply want to change the set of data they gather.

There has also been work relating to extending sensorboards to handle a large variety of devices. These include the Extendible Sensing System (ESS) [21] and the MTS300 multi-sensor module [23]. ESS is a multi-tier system that addresses sensor collection and publication. ESS nodes are divided into two sets. One is the often resource poor set of sensor nodes. The other is the set of gateway nodes designed to manage sensor nodes. ESS also addresses the issue of data publication and gives end users the ability to subscribe to data streams from a deployment. The MTS300 multi-sensor module is a sensorboard from Crossbow that integrates sensors such as light, temperature, and sound. Both of these, however, are designed for the Mica platform, and do not address our requirement of using the Monibus sap flow sensor and flexibly retasking it.

7 Discussion

We now discuss the lessons we have learned in building sensor network system for long term deployment using a set of existing services. Much of our discussion is about struggles with building the Savia system and suggestions for how we might improve the OS for sensor networks and their underlying platforms.

7.1 Resource Arbitration

Resource arbitration was a critical issue during our development of the Savia system. Misconfigured hardware resources often led to non-deterministic conflicts that were extremely difficult to resolve. In our system, we had to explicitly serialize the initialization of the mote, so that the radio initializes either before or after the flash, but not at the same time. We also had to turn off the radio component because of the asynchronous nature of the Monibus sensors. While the Monibus is in use as a UART, the Telos bus pins should atomically stay in UART mode. However, if the radio is on and a message is received, it will trigger and switch the data bus back into SPI mode.

As more and more mote platforms are used to attached customized sensors, it becomes more important that each device is able to properly share the resources. TinyOS currently has such components like BusArbitrationM to exclusively get and release the bus, but it puts the onus on the application developer to use it properly. Protection between resources will also be increasingly important so that improper flash accesses will not leave the radio in an unusable state.

7.2 Platform Resources

Our experience with Savia also shows that program size increases faster than memory size as shown in Figure 6. This is because as more components are added, more program logic must be added to handle it. In addition to code for handling the basic functionality, code must also be added to integrate properly with the rest of the system. In Savia, the bus arbitration code had to be written in addition to the basic functionality code. We also had to augment the synchronized timer code to filter out invalid readings. All this code added up in ROM as opposed to RAM.

In early hardware platforms, RAM was the limiting factor. However, with RAM already in the 10K range, ROM is now the limiting factor. In Savia, we did not even use half the available memory in building a complex system. As sensor network deployments have more complicated program logic, more ROM space will be necessary to accommodate these systems. We could have customized the application and built a lean monolithic application for use with a specific deployment, but we would have lost our simple retasking capability. Thus to satisfy our requirements, one of our design goals was to build a system general enough for different sensing needs in varying environments.

7.3 Power Management

Power management is an important consideration for wireless sensor networks, since many network devices rely solely on a limited supply of battery power. Although we make use of attached solar panels to replenish the devices, we must still minimize power usage - to reduce the size and cost of panels, and to ensure that the devices do not depend on fine weather to operate properly.

The key principle behind power management is simple - ensure each component is in its lowest possible power state when not in use. In many cases the existing hardware and drivers meet this requirement. The temperature/humidity sensors and the non-volatile storage both revert to an ultra low power sleep state when not in use (< 50 uA). However, the radio remains in receive state to hear incoming messages (such as code, sync beacons, routing data), and as such draws a significant amount of power.

Traditional operating systems generally have more than adequate energy supplies, and thus do not need to provide an energy aware programming interface. However, with sensor networks designed to operate for long periods of time and with limited energy supplies, it is imperative that energy aware constructs be exposed to the application developer.

8 Conclusion

Building complex sensor network systems out of heterogeneous hardware components and existing services is hard not necessarily because of the nature of their operations, but in the integration of these components into a complete working system. Hardware resource sharing poses unique challenges to building these systems, which should be addressed not at the application layer, but at a service layer below. Building systems vertically has been the traditional way of creating new sensor network applications, but it has caused the reinventing of many wheels. Reusable components has allowed traditional software to be rapidly developed and deployed. Sensor network applications can also benefit from this. We have developed Savia, a sensor network system for measuring the sap flow in redwood trees based on these principles. Although there were challenges in integration, the realm of resuable components in sensor networks is promising.

References

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

[2] S. Burgess and T. Dawson. The contribution of fog to the water relations of sequoia sempervirens (d. don): foliar uptake and prevention of dehydration. In *Plant Cell and Environment*, 2004.

[3] Crossbow, Inc. Mpr2400: Micaz zigbee series. http://www.xbow.com/Products/productsdetails.aspx?sid=62, 2005.

[4] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, 2005.

[5] J. Eastham, C. Rose, D. Cameron, S. Rance, T. Talsma, and D. C. Edwards. Tree-pasture interactions at a range of tree densities in an agroforestry experiment ii. water uptake in relation to rooting patterns. In *Australian Journal of Agricultural Research*, 1990.

[6] C. O. et al. Biodiversity meets the atmosphere: A global view of forest canopies. In *Science*, 2003.

[7] D. Gay. Design of matchbox, the simple filing system for motes, 2003.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.

[9] R. Govindan, E. Kohler, D. Estrin, F. Bian, K. Chintalapudi, O. Gnawali, S. Rangwala, R. Gummadi, and T. Stathopoulos. Tenet: An architecture for tiered embedded networks. Technical Report CENS-TR-56, November 10 2005.

[10] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proceedings of the Seconnd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[11] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[12] T. Hatton, S. Moore, and P. Reece. Estimating stand transpiration in a eucalyptus populnea woodland with the heat pulse method: Measurement errors and sampling strategies. In *Tree Physiology*, 1995.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[14] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[15] 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 *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, oct 2002.

[16] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[17] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.

[18] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, 2005.

[19] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *Transactions on Database Systems (TODS)*, 2005.

[20] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, Nov. 2004.

[21] E. Osterweil, M. Mysore, M. Rahimi, and A. Wu. The extensible sensing system. http://research.cens.ucla.edu/pls/portal/url/item/F2EC99597DC1942DE0306 2003.

[22] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *In Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April. 2005.

[23] M. Rahimi. The mica2 data acquisition board. http://www.cens.ucla.edu/ mhr/daq/datasheet.pdf, 2003.

[24] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[25] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[26] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of Second European Workshop on*

Wireless Sensor Networks (EWSN 2005), 2005.

[27] G. Tolle, J. Polastre, R. Szewczyk, N. Turner, K. Tu, P. Buonadonna, S. Burgess, D. Gay, W. Hong, T. Dawson, and D. Culler. A macroscope in the redwoods. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.

[28] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.