# TASK: Sensor Network in a Box

Phil Buonadonna*, David Gay*, Joseph M. Hellerstein*‡, Wei Hong* and Samuel Madden†

*Intel Research Berkeley
{pbuonado, dgay, whong}@intel-research.net

‡UC Berkeley
jmh@cs.berkeley.edu

†MIT
madden@csail.mit.edu

***Abstract*— Sensornet systems research is being conducted with various applications and deployment scenarios in mind. In many of these scenarios, the presumption is that the sensornet will be deployed and managed by users who do not have a background in computer science. In this paper we describe the "Tiny Application Sensor Kit" (TASK), a system we have designed for use by end-users with minimal sensornet sophistication. We describe the requirements that guided our design, the architecture of the system, and results from initial deployments. Based on our experience to date we present preliminary design principles and research challenges that arise in delivering sensornet research to end users.**

## I. Introduction

A standard vision of wireless sensor networks involves an end-user buying a collection of sensor nodes, powering them up, and sprinkling them – literally or figuratively – within an environment. The devices automatically form an ad-hoc network, sense their environment, and report readings back to a central location over the course of months or years. In these early years of sensor network technology, things are not yet that simple. Most deployments today happen via a squad of computer science researchers working hand-in-hand with a potential user to carefully deploy nodes, configure and even write software on the fly, observe network behavior and analyze incoming data, and return on a regular basis to monitor and maintain the health of the system [1].

Over the past two years, we have been working on the Tiny Application Sensor Kit (TASK), a "turnkey" sensornet application for Berkeley motes. TASK is intended to encourage sensornet adoption by making environmental monitoring deployments relatively self-explanatory, easy to configure, and easy to maintain. Our work on TASK was guided by our initial deployments of sensornets in partnership with the scientific [2] and agricultural [3] communities, as well as initial discussions with industrial partners focused on HVAC [4], equipment monitoring [5] and asset management [6].

These experiences have helped us identify a set of user requirements for TASK (Section II), and helped guide its design and implementation (Section III). We believe that these requirements are not specific to TASK but are necessary for any similar environmental monitoring system. TASK is not yet fully mature, but early experiences with a couple of deployments (Section IV) show that our basic architecture is sound and have taught us a number of lessons (Section V) which should help TASK's future development. Our overall experience is that deploying sensornet applications in the real world is significantly harder than laboratory tests and simulators would indicate. We believe that our efforts to build a turnkey sensornet kit are an important step in addressing critical sensornet challenges in a meaningful manner.

## II. User Requirements

A large variety of sensor network applications have been proposed, from environmental monitoring to structural monitoring to asset tracking. Clearly, it is impossible to design a single sensor kit to satisfy the requirements of all possible applications. Thus, we decided to focus our efforts on building a kit for low data rate, environmental monitoring applications. Although we have since worked on extending TASK for new types of applications, they are beyond the scope of this paper. We derived the requirements for TASK both by talking to many potential sensor network users with environmental monitoring applications, and by working with some of them closely in real sensor network deployments [2].

### A. Design Requirements for End Users

End users of sensor networks are typically experts in their own fields such as biology or agriculture, but often not sophisticated computer users. Thus, one of our primary goals was to alleviate the need for end-users to program devices or "babysit" a network.

With this intuition in mind, we laid out initial requirements for TASK, which we revisit throughout the paper:

EU-1 **Ease of software installation**. The installation of the system software on any device (including

standard desktop PCs) must be extremely simple.

EU-2 **Deployment tools**. After software installation, users are faced with the task of placing sensor nodes at required sensing locations, deploying routing nodes to ensure good network connectivity, and establishing connections between the gateways and the Internet. To the extent possible, users should be aided in their deployment efforts by tools that provide network and data quality feedback. Users should also be able to add nodes to a deployment with a minimum of effort and plenty of feedback.

EU-3 **Reconfigurability**. Sensor network users often need to reconfigure their applications over the course of a deployment to fine-tune data rates or types of data collected, or to adjust network size.

EU-4 **Health monitoring**. Users need tools to verify that all sensor nodes are connected to the network and reporting meaningful data.

EU-5 **Minimum network reliability guarantee**. Most sensor network users expect some noise and data loss from sensor networks. Extensive loss, however, is unacceptable.

EU-6 **Interpretable sensor readings**. To end users, raw sensor readings are meaningless. They must be able to receive and manipulate sensor readings in well-known engineering units.

EU-7 **Integration with data analysis tools**. A sensor kit should use standard data formats (comma delimited text, ODBC/JDBC interfaces, etc) that permit access by a variety of analysis tools.

EU-8 **Audit trails**. Sensor networks may exhibit unexpected behavior – catastrophic losses, delayed transmission, node failures, and so on. Maintaining a record of every human manipulation of the network and every single packet that flows out of the network will help diagnose such unexpected faults.

EU-9 **Network longevity estimates**. There must be a method to provide some reasonably accurate estimate of the network lifetime based on the current sensornet configuration. For example, the biologists monitoring the bird habitat on Great Duck Island [1] need the network to last for at least 6 months as that is the duration of the nesting season on the island.

### B. Design Requirements for Developers

There were additional requirements necessary to make TASK a flexible system for developers. These included:

D-1 The need for a **familiar system API**, allowing the easy development of new real-time data visualization or network monitoring tools without rewriting portions of TASK.

D-2 The need for **extensibility of sensor node software**, for adding new types of sensors or new data transformation operators.

D-3 **Modular in-network services**, to facilitate experimentation with different solutions for routing, sampling, scheduling, etc.

## III. TASK Design

In this section, we describe the design and implementation of TASK. We start with a general discussion of our design principles followed by a description of the overall architecture and its key components.

### A. Design Principles

The driving principles we adopted in designing TASK are *simplicity over functionality*, *modularity*, *remote controllability* and *fault tolerance*. In this section we consider each principle in turn, and highlight some its key influences on TASK.

**Simplicity over Functionality.** Many of the components needed to build a general purpose sensor kit have already been developed by the sensor network community. For this reason, we decided to build TASK based on the "best of breed" existing components already developed by the TinyOS community. Our efforts have focused on hardening existing technology, integrating it into a single package, and evaluating its performance in real world deployments. When functionality had to be implemented from scratch, we tended to use the simplest possible solutions. For example, we struggled for several months to integrate a general purpose, high precision time synchronization layer into TASK – it consistently caused nodes to crash and become desynchronized under high load. The better approach was a simple, application-specific approach that achieves millisecond accuracy and is absolutely sufficient for our needs. (Section III-C.)

**Modularity.** Though modularity is key to any good software engineering practice, it is particularly important in using a best-of-breed approach for emerging technology, because the components will evolve over time and will need to be replaced. With this in mind, we structured TASK as a set of loosely-coupled layers. The client applications/GUIs access the TASK server (Section III-D) over a simple interface. This server communicates with the sensornet via a narrow, well-specified abstraction that permits easy substitution of different sensornet data collection models. The core in-network data collection software is separable from the

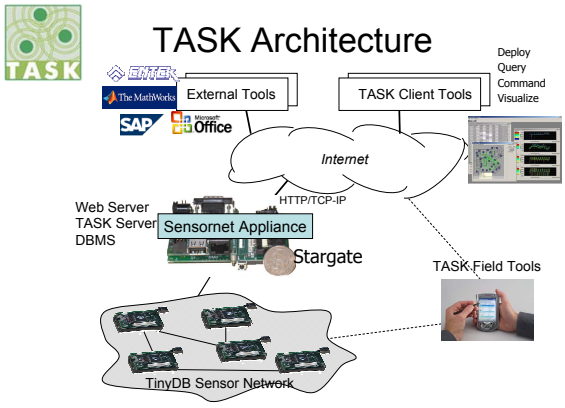


Fig. 1. The TASK Architecture

communication provider allowing for different multi-hop protocols and radio power modes (see Section V.)

**Remote Management.** It is frequently desirable to interrogate and control remote nodes throughout a sensor network deployment. Nodes tend to be installed in hard-to-reach or remote locations so significant time and effort can be saved if nodes can be remotely managed.

**Fault Tolerance.** Failures are much more common in sensor networks than traditional IT systems because of cheap hardware components and challenging environments. We took a simple approach to fault tolerance: whenever a failure is detected, nodes restart, retaining enough state (or inferring enough state from other, non-failed nodes around them) to continue data collection.

### *B. System Overview*

Figure 1 shows the overall architecture of TASK, which is covered in detail in the rest of this section. A TASK sensor kit consists of a collection of sensor nodes (Section III-C), a *sensor network appliance (SNA, Section III-D))*, a number of TASK *field tools* running on PDAs, and TASK *client tools* running from any computer connected to the Internet (Section III-E). In the spirit of the end-user design requirement EU-7, TASK also integrates easily with popular data analysis tools such as Matlab, Excel, Labview, etc. through standard database and text export interfaces.

### *C. Sensor Nodes and Software*

The TASK sensor nodes are `mica2` or `mica2dot`, running TinyDB [7], a distributed query processor for TinyOS motes. TinyDB presents a sensor network as a virtual database table called *sensors* consisting of all the attributes defined in the network on all of the sensors. This database is queried using a SQL-like query language called TinySQL. TinySQL is a slightly modified version of SQL. The main difference is that TinySQL introduces a new "SAMPLE PERIOD" clause to SQL's "SELECT ... FROM ... WHERE ..." syntax. The "SAMPLE PERIOD" clause specifies how frequently the sensors are sampled to generate a tuple in the virtual table. Alternatively, the sample period can be derived from a lifetime requirement using the "LIFETIME" clause, where the sample period is computed from the lifetime goal using a simple model of a query's energy consumption. An example sensor query is:

```
SELECT nodeid, temperature, humidity
 FROM sensors
 SAMPLE PERIOD 5 min
```

TinyDB supports multiple, concurrently running queries, and a command interface for controlling other aspects of mote behavior. TinyDB is relatively mature, and satisfies many of Section II's requirements. A data collection query collects the user's data, thus supporting reconfigurability (EU-3), and, via the "LIFETIME" clause, longevity management (EU-9). A separate health query collects attributes representing network topology (EU-2) and system parameters (EU-4). TinyDB allows developers to extend the system with new data transformations, and with new *attributes* to query different types of sensor or access internal metadata (Developer Requirement D-2). The interfaces between TinyDB and TinyOS are also sufficiently modular for our needs, as required by D-3. For instance, it works over a variety of multi-hop routing implementations.

However, TinyDB was not ready for real-world deployment when we began our work. We made the following changes to TinyDB to facilitate real-world deployments: we added simple power management and time synchronization features, a *query sharing* feature, a watchdog, and on-mote logging. We discuss these improvements in turn.

**Power Management**. TinyDB initially left nodes on all the time, which limited lifetime to just a few days. To meet TASK network lifetime requirements (EU-9), we needed to add power management features. We have explored two power management solutions for TinyDB: *duty cycling* and *low-power listening*. In duty cycling, nodes are awake for a short, synchronized period each sample interval. This period must be long enough for a node to sample sensors, send its data and forward data for its children. Ideally, this period would be adaptive (e.g., see [8]), but currently TinyDB uses a fixed period (selected for each deployment, typically 4s). In low-power listening, each node wakes up periodically to sample the channel for traffic and goes right back to sleep if there is nothing being sent. In this mode, nodes

send messages with a preamble that is at least as long as the period with which listeners sample the channel to make ensure that the messages will be received by the destination nodes. The advantage of low-power listening is that the TinyDB application layer no longer needs to manage the scheduling of wake and sleep. The disadvantage is that is substantially increases transmission cost. A preliminary performance comparison of these two approaches is presented in Section IV-C.

**Time Synchronization**. Time synchronization is required for TinyDB to schedule nodes to wake up at the same time. Even with low-power listening, time synchronization is desirable because it ensures that readings from sensors across different nodes were taken at the same time. TinyDB uses the approach proposed in many research papers [9], [10] of time-stamping every packet just before it is transmitted. When a node overhears a packet sent by its parent, it synchronizes its time with its parent's. Thus, the entire network is eventually synchronized (in a step-by-step fashion) to the time of the root node. When a node does not hear from its parent for a number of sample periods, it will stay awake for an entire sample period to re-synchronize with its parent node. This simple approach achieves an accuracy of 1-2 milliseconds in our real deployments, which is sufficient for purposes of our scientific users and our scheduled communication protocol.

**Query Sharing**. Query sharing is a mechanism for reliable query dissemination in TinyDB. It also allows new or restarted nodes to learn about currently executing queries, which is important both for network evolution (EU-2) and fault tolerance. It works as follows: When a node overhears a neighboring node transmitting a data packet, it checks the 8-bit query id embedded in the packet against the ids of the queries that it is executing. If the query id is unknown, it sends a query request message to the source of the overheard data packet. When a node receives a query request message, it broadcasts the query corresponding to the requested query id. This way the node missing this query will receive and begin executing the query. Our initial implementation of query sharing did not scale to a large number of nodes: whenever a node heard a result for a missing query, it would send a query request, and every node that heard that request would reply. This resulted in a huge amount of traffic. To see this, consider a network with $n$ nodes in a single cell, where $n/2$ of them (the *"starters"*) know about the query, and the other $n/2$ (the *"others"* do not. A TinyDB query is transmitted in $k$ messages, typically for some $k > 1$. When each of the *starters* sends its query results, each of the *others* will send a query request to each of the *starters*, who will respond with a $k$-packet response. Thus, $(n/2)^2 * (k+1))$ messages are generated in a single epoch. For $n = 10$ and $k = 5$, this is already 150 messages. Such traffic loads cause the TinyOS MAC to collapse. We implemented the following optimizations to fix this problem:

- Receive bitmap. To reduce the size of the query messages, we added a bitmap indicating which parts of a query are needed to the query request message. The receiver of the query request only sends out those needed portions.
- Backoff. To prevent duplicated transmission of a query, if a node hears one of its neighbor nodes request the same query, it suppresses its own request.
- Rate limiting. We no longer allow a node to send a query request message more than once per query sample epoch.

**Watchdog**. TinyDB uses the watchdog component in TinyOS to achieve two purposes: fault tolerance and remote controllability. At the start of every $k$ sample periods, TinyDB starts the watchdog. The watchdog will restart the node if TinyDB does not reset the watchdog before it expires. The watchdog is reset whenever the node hears a message. Thus, the watchdog will restart the node if it does not hear any messages for $k$ sample periods. After a node is restarted, it obtains the current TinyDB queries via query sharing. This approach also ensures that the node is always reachable remotely because if the node's radio stack crashes, the watchdog will trigger.

**Logging**. We added an optional "data logger" to TinyDB to preserve all query results, in the spirit of EU-5. The goal of this component is to provide extra durability, by allowing recovery of historical readings when the motes are brought back from the field. Clearly it is not intended as a substitute for live data, but rather as a backup technique.

The data logger uses the `mica2` 512kB flash chip and a "persistent logger" component that provides reliable, record-oriented logging. Each record contains the mote's current time and the data portion of a TinyDB message. In the current implementation, this record takes 31 bytes, so we can save the first 16500 query results which is sufficient for short deployments (e.g., this is 57 days at one query result every five minutes).

The persistent logger supports only three operations: erase the log, append a new record, read the log. It is designed to provide reliability in the presence of failures such as power loss or sudden reboot in the following

sense: after recovery, the log will be readable up to the end of some record (i.e., it isn't possible to read a partial record), and logging will resume at this same point. Note that this still allows some amount of data loss at each failure.

The implementation is fairly straightforward: The flash is divided into blocks (256 data + 8 metadata bytes) which can be individually written; we assume that these blocks are the unit of failure. Records are written consecutively into the data portion of these blocks, and may straddle block boundaries. The metadata stores a cookie, the offset of the last record that ended in this block (if any), and a CRC (on the data+metadata). At boot time, the end of the log is found by performing a binary search for the last valid block.

We a used binary search to avoid placing any global metadata (e.g., last write offset) at a fixed location, as flash chips only allow a limited (e.g., 10000) number of writes to any one address. Our binary search assumes that the flash contains a sequence of valid blocks followed by invalid blocks. This covers the failures we are concerned with, but will fail if, e.g., a block becomes corrupted after the next block has been fully written.

### *D. The Sensor Network Appliance (SNA)*

The Sensor Network Appliance (SNA) serves as a portal between a sensor network and the clients. The SNA is a self-contained base station that requires little setup other than a network connection and power source (EU-1). It uses a radio compatible with sensor nodes and provides local data storage and management functions for the sensor nets associated with it. It also includes wide-area connectivity via a persistent Ethernet connection (when available) or a satellite or cellular connection in remote settings. The SNA includes a DBMS that provides local storage for data results, health monitoring and logging as required by the TASK Server. The DBMS is accessed via a standard ODBC interface. Additionally, the TASK Server (see below) provides an HTTP interface accesible via a browser or directly by applications.

While a laptop could have served as our SNA, they do not typically have a desirable size or power footprint, and are difficult to harden against the environment. Instead, we adopted the Stargate platform from Intel Research and Crossbow [11]. The Stargate is a palm-sized, general-purpose computer that houses a 255Mhz Intel xScale processor, 64 MB of memory, and a provides variety of connectivity options including PCMCIA, Ethernet, USB, RS232, and a native connector for the mica series of sensor motes. It can be powered by batteries. The Stargate runs the Linux operating system and supports the full range of software tools available for this environment. Updates to the software can be made by simply inserting a new compact flash card.

| *Method* | *Description* |
|---|---|
| `runQuery` | Creates the necessary DBMS table and issues the given sensor query. |
| `addListener` | Add a callback method for certain types of query results. |
| `runCommand` | Issues the given command to the sensor net. |
| `stopQuery` | Stops a given query if it is running. |
| `getQuery` | Retrieves the running sensor query. |
| `addCliInfo` | Creates a new configuration. |
| `deleteCliInfo` | Deletes a configuration. |
| `addMote` | Adds a sensor to a configuration. |
| `deleteMote` | Deletes a sensor from a configuration. |
| `runDBMSQuery` | Executes a result query on the SNA DBMS. |

TABLE I

TASK SERVER ABSTRACT API

The TASK Server implements the core functionality of the SNA. Externally, the server exports an interface to clients for submitting sensor queries and commands, monitoring health and browsing DBMS data (see Table I). On initial startup, the TASK Server initializes TinyDB, the internal web server and the local DBMS. Prior to issuing a sensor query on a new network, the client creates a sensor network *configuration* which is a persistent logical grouping of nodes with attached metadata such as the location or attached sensors. This configuration helps manage related sensor nodes on the same physical network, and provides an affordance to users about previous deployment decisions (EU-2).

Once a configuration has been selected (or created), the server listens for incoming queries or commands and handles data result arrivals. When a client issues a query, the server logs the query to the DBMS (EU-8), creates a new result table for the query in the DBMS and then passes the query on to the sensor network software (i.e., TinyDB) for processing. Multiple queries may be issued to the sensor network simultaneously. When results from a sensor query are received, the TASK Server stores the results in the SNA DBMS and then sends the data to any active clients that are connected. Alternatively, the server can periodically replicate results to an external database. In the TASK architecture, the SNA DBMS is not necessarily the stable store for sensor results. Rather, it is intended to be a flexible staging buffer in front of an external database or client tool.

We implemented our TASK Server in Java. It provides a sockets-based client interface as well as an HTTP front end via the Jetty HTTP server [12]. The implementation required approximately 4800 lines of source code.

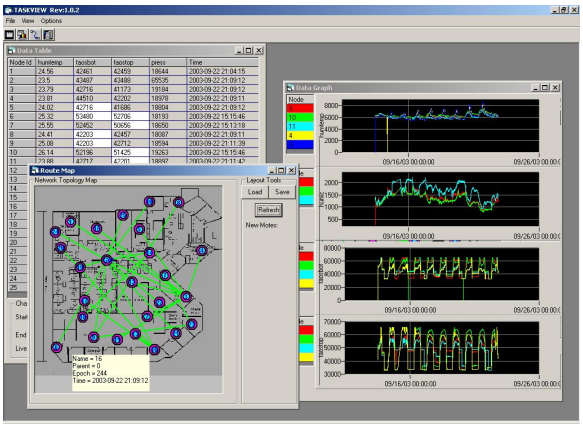

Fig. 2. The TASK View Client Tool

### E. Tools for the User

TASK comes with two client tools: a simple web-based tool for interacting with the sensor network through the SNA and a full-featured Visual Basic-based tool called *TASK View* that provides more advanced features such as node deployment bookkeeping, network visualization, and near-real-time continuous monitoring. Figure 2 is a sample screen shot of TASK View. The lower left window shows the deployment and configuration screen, while the two other windows show a tabular and graphical view of data from a set of motes.

The field tool is a simple tool for debugging TASK-based sensor networks "in the field" (EU-2). The field tool allows a user to walk up to a TASK-mote and inspect or modify its state by issuing a set of simple commands. The two primary commands are "ping", to query the mote's network status (current query, parent) and "reset" to reset it. The field tool only operates on the motes that are in direct radio range (i.e., close proximity) to the user. The field tool runs on handhelds devices (we currrently use Sharp's Zaurus handheld) equipped with a `Canby` or `Canby2` compact-flash mote.

The field tool broadcasts periodic beacons asking motes to report in so that it can provide the user with a list of the local motes. When TinyDB is used in duty-cycling mode, this may take a while (up to a whole sampling period). Once a mote notices these beacons it disables duty sampling to allow interaction with the field tool (duty cycling is re-enabled when the beacons are no longer heard).

When many motes are in the proximity of the field tool, responses to the periodic beacon can saturate available radio bandwidth. To avoid this, motes respond to beacons after a random delay, and suppress their responses if they overhear more than $n$ other motes (currently, $n = 5$) or if they have responded recently. The combination of periodic beacons, random delay and duplicate suppression allows the field tool to provide an accurate list of motes even when their density is high.

## IV. TASK Deployment Experiences

In this section, we study the details of our two most significant TASK deployments to date: a 54 node deployment in our lab (hereafter, the lab deployment) that ran for about a month, and a 23 node deployment in the UC Berkeley Botanical Garden (hereafter, the garden deployment) that ran for about 20 days. We also performed a more controlled study in our lab (Section IV-C), to compare the behavior of three routing layers: the original, standard TinyOS routing used in the first two deployments; MINTRoute [13]; and a variation on MINTRoute using a low-power radio stack [14].

The goal of these studies is to validate the use of TASK for long-lived, low data-acquisition rate applications, not to perform a detailed comparison of various routing approaches. We want to ensure that we can gather data at a sufficient rate for enough time, so we measure yield (percentage of requested data received) and average current consumption.

### A. The Lab Deployment

Figure 3 shows a map of the 54 node deployment in our lab. Data was routed towards node id 0 and the furthest nodes (in the upper left corner of the deployment) were about 10 hops from the root. For example, node 51's most common route is 52, 54, 9, 10, 11, 14, 18, 24, 26, 0. Sensors collected light, temperature, humidity, voltage, and network topology information every 30 seconds. The nodes were active during March, 2004 and lasted between 28 and 31 days (about $30 \times 24 = 720$ hours), using the scheduled communication approach described in Section III-C, with a waking period of four seconds (a 4/30=13% duty cycle). Four seconds reflects a one-second sensor start up time, plus a three-second window during which motes deliver their data over the network. Choosing this value properly is critical to both the longevity and quality of data collected in a deployment, an issue which we discuss in more detail in Section IV-C.

The deployment served two purposes; first, it provided a way for us to verify that our power management and watchdog facilities could keep a network running unattended for long periods of time. Second, it gave developers of sensor network applications and algorithms a data set to experiment with.

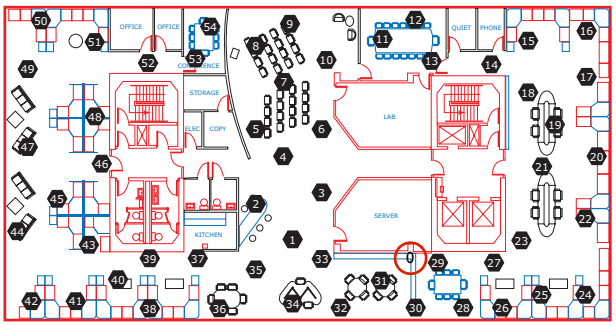


Fig. 3. A map of our lab, with mote locations labeled in black.

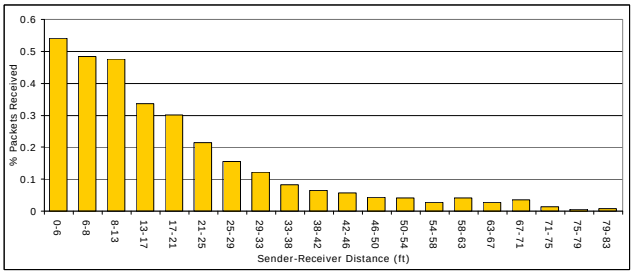


Fig. 4. Percentage of received packets versus distance between sender and receiver (in our lab).

We added support for a special "network neighborhood" attribute to TinyDB prior to this deployment. This attribute consisted of a 64-bit bitmap where the $i$th bit in $j$'s bitmap on epoch $e$ indicated whether or not $j$ had heard node $i$ transmit on epoch $e-1$. This provided a way to study the network graph in detail and has proven a to be useful tool in a number of network design scenarios. Furthermore, we believe this is one of the first networking data sets to characterize loss in a high-contention environment, as opposed to a radio-controlled environment where only a single sender and receive are active at a time, as is common in the literature [13], [15]. Figure 4 shows a histogram of the reception rates versus distance in feet computed using this data set; we omit a complete study of this data as this is not the main focus of this paper.

We also measured the percentage of total results received from each of the nodes in the deployment. The results are shown in Figure 5. Nodes 5, 8, 15, and 51 failed after the first few days; the cause of failure is unknown. Notice that most nodes get substantially less than 75% of results, with some getting as few as 30%. Interestingly, it is nodes 11-20 that perform the worst; based on anecdotal evidence, we have reason to believe that this corner of the lab is subject to unusual interference.

Losses do not appear to be particularly correlated across sensors, though they are certainly highly correlated within a single sensor; Figure 6 shows losses for a 1 hour period on sensors 1-10. Notice relatively large gaps on individual sensors that do not usually correspond to gaps on other sensors. We observed similar patterns throughout the data set.

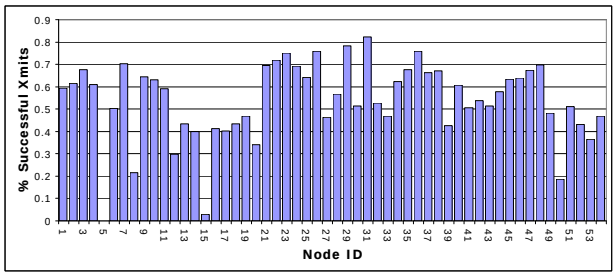


Fig. 5. Percentage of successful transmissions by sensor (in our lab).

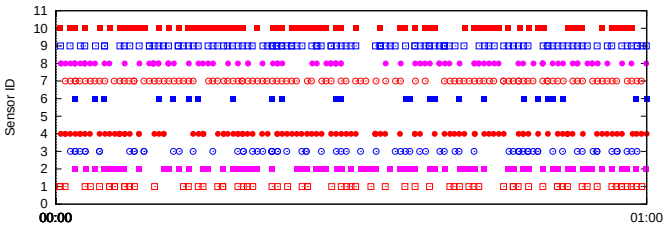


Fig. 6. Distribution of losses by sensor, during a 1 hour period (in our lab).

The large percentage of losses in this deployment, as well as in our initial redwood deployment, led us to evaluate techniques for mitigating loss. These efforts are described in more detail in Section IV-C below.

### B. The Garden Deployment

The garden deployment consisted of 23 motes running for about 20 days (480 hours) during the summer of 2003 at a sample period of 30 seconds. In this case, motes were deployed in special weatherproof packages with 800 mAh lithium-ion batteries. Motes sensed photosynthetically active radiation (PAR), humidity, temperature, and barometric pressure.

Devices were placed in a 34m redwood tree in the UC Berkeley Botanical Garden at four different altitudes. The goal of the deployment was to measure how environmental parameters varied throughout the day at different heights in the forest canopy. Such variances can have a significant effect on the models used to predict the growth of trees [16]; for example, variations of just 20% in humidity (either up or down) can reduce tree growth from a normal rate to almost nothing. Such models play a critical role in forestry management because incorrect predictions can have a significant negative economic

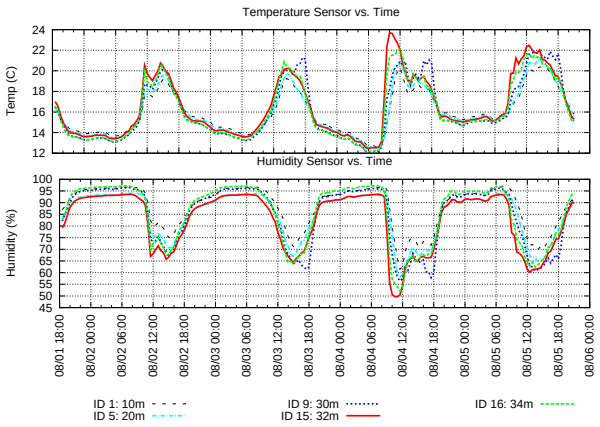


Fig. 7. Temperature (top) and Humidity (bottom) Readings from 5 Sensors in the Botanical Garden

impact on the logging industry.

Motes radioed data from the tree, which was in the middle of a small grove, back about 100m to a building where we had installed a large antenna and the TASK base station which collected results on a local database. Due to the range of our antenna the network was mostly single hop, although motes on the far side of the tree needed several hops.

For power management purposes, motes were configured to use a waking period of 2 seconds (as discussed below, we determined that this period was too short, and switched to a longer period of 4 seconds in the subsequent lab deployment), so the active duty cycle was 2/30 = 6.6%.

*1) Result Summary and Data Quality:* Figure 7 shows data from five of the sensors collected during the first few days of August, 2003. The periodic bumps in the graph correspond to daytime readings; at night, the temperature drops significantly and humidity becomes very high as fog rolls in.

This variation in humidity and temperature throughout the canopy is one of the effects the biologists involved in the project are interested in studying. In particular, they wish to understand how textbook models relating tree growth to temperature, humidity, and light should be changed to accomodate these intra-canopy variations, since these existing models are based on simple coarse-grained characterizations of temperature and humidity that vary significantly with the general weather. However, as we see in our data, the trees act as a buffer, releasing water into the air on hot days and absorbing it on foggy days. Looking closely at the line for sensor ID 1, it is apparent that not only is the mote cooler and wetter during the day, but at night it is warmer and less humid than the exposed sensors near the treetop. Furthermore, the rate of change of humidity and temperature at the bottom of the tree is substantially less than at the top – Sensor 1 is the last sensor to begin to warm up *and* the last sensor to cool off. Given these measurements, our colleagues have argued that earlier growth models take a simplistic view of a forest's microclimates – one that does not accurately capture the dynamics across the forest's three-dimensional volume [2].

Perhaps the most significant result here is the qualitative change in measurement ability afforded by the sensor net's fine-grained readings. The density of measurement provided by even these early TASK deployments changes scientists' ability to measure and understand natural processes.

We also studied the loss rates in this deployment. Figure 8 shows the number and percentage of results received from sensors 1-16, with a breakdown by parent in the routing tree. Sensors are along the X axis, and the packet count is along the Y axis. Shading indicates the portion of each sensor's packets that were forwarded by a given parent. Parent ID 0 is the base station – note that the majority of readings for most sensors are transmitted to this node in a single hop. Only sensors 6,7,9,11 and 14 transmit a significant portion of their packets via multiple radio hops. It is interesting to note that these are sensors in the middle of the tree – nodes at the bottom and very top seem to be better connected.

In this case, the best sensor (ID 2) successfully transmitted about 75% of its data. Except for sensor 3 (which failed), the worst performing sensor, 14, successfully transmitted 22% of its results. Losses are uniformly distributed across time and do not appear to be correlated across nodes.

We suspect that the primary cause of loss is network collisions. Despite the fact the each node transmits infrequently, time synchronization causes sensors wake up and transmit during exactly the same 2s communication interval. Even without multi-hop communication, this means that at least 16 packets are sent during the same second, which is difficult for the TinyOS MAC layer to handle. This observation led us to switch to a 4 second backoff period in the lab deployment. We have yet to analyze this hypothesis in detail, in part because of the shift to 802.15.4 radios in products like the `MICAz`, which changes a number of aspects of the communication layer, and appears in our early experiences to provide much lower loss rates.

*2) Calibration:* Calibration is essential to provide users with numbers in well-known units that they can

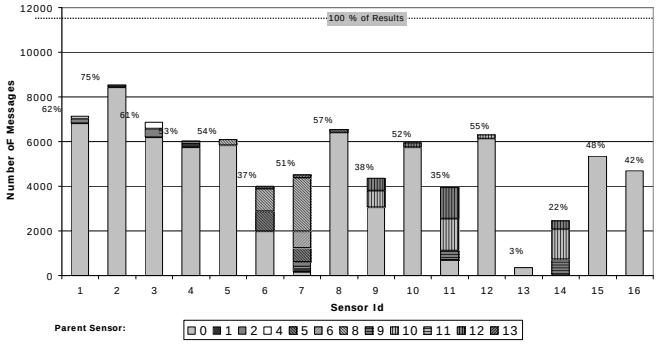


Fig. 8. Graph showing the number of results received from every sensor, broken down by parent in the routing tree.

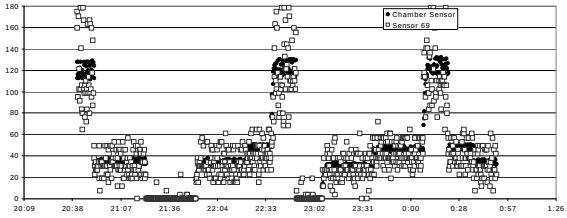


Fig. 9. Comparison of readings from a manufacturer calibrated light sensor on motes and high-cost calibrated sensor in an environmentally controlled chamber.

understand and integrate into their existing models and software. It is also one of the trickiest and least anticipated issues that arose with the garden deployment. We had selected sensors that we believed had been reasonably calibrated before they left the factory [17], [18], [19]. However, when we began comparing the output of these sensors to much higher cost sensors that had been calibrated by the biologists we were working with, the results we disappointing. As an example, Figure 9 compares the value from the light sensor we had selected (calibrated according to the manufactured specification) and a calibrated sensor in an environmentally controlled chamber – although the two sets of readings are correlated, the values between the two sensors at the same time often vary by a large amount.

In this case, we were able to produce readings that much more closely matched the calibrated sensor by reporting the value at each time as the median of 10 readings from the previous 10 seconds. This had the effect of smoothing the curve and eliminating much of the noise, but also decreased the responsiveness of the sensor and consumed significantly more energy. Figure 10 shows the results after the median had been applied.

Our current calibration protocol calls for comparing each sensor on each mote against a well-known baseline (as in the calibrated chamber setting) to verify it is

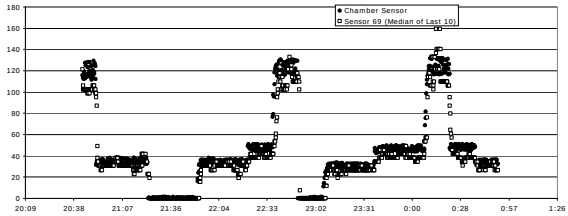


Fig. 10. Same data as in Figure 9, except that each point is the median of ten readings from the previous 10 seconds; this has the effect of removing noise from the signal, producing a result that matches much more closely with the calibrated chamber sensor.

behaving correctly and to establish the values of per-sensor scaling parameters. This protocol also demands that we re-measure sensors versus this baseline after a deployment is complete, to verify that the passage of time hasn't affected a device's ability to sense.

In general, calibration is tricky because the appropriate calibration function varies from sensor to sensor. For example, many of our sensor's readings are affected by the voltage of the device, but the exact relationship varies across different sensor types. This makes deriving appropriate calibration functions a time-consuming and ad-hoc process. Unfortunately, proper calibration is an essential feature for anyone using these devices for precision monitoring, as in many scientific and industrial environments. We see this as one of the primary challenges facing TASK– and sensor networks in general – as we seek wider adoption. Calibration has received scant attention in the sensor network research community, but is a critical aspect in the practical use of the technology. We encourage more research on this topic.

### *C. Performance Benchmarks*

To help understand the network reliability and power consumptions results observed in our deployments, we conducted several controlled, small-scale experiments. Three different routing and radio driver configurations were used to better understand the impact of the communication layer on these parameters. The first configuration used the standard routing layer in TinyOS 1.1 (located in tos/lib/Route) and the default radio stack for the MICA2 platforms. This routing layer implements a distance-vector protocol using shortest path to the root and link quality of the next hop [13]. The second configuration utilized the minimum transmission routing provider (MINTRoute) which chooses a path to the root based on the aggregate quality of a links along different paths [13]. The final configuration used a variant of MINTRoute coupled with a low power radio stack [14].

For each of the configurations, queries with sample

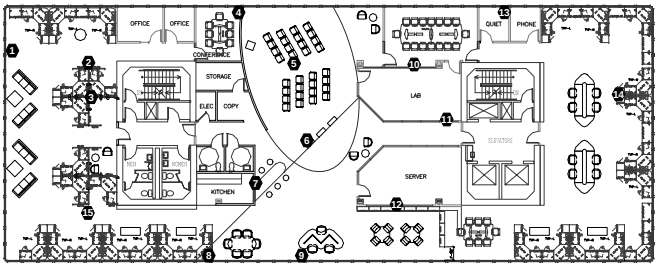


Fig. 11. In-lab 15-node sensor network used for performance benchmarks.

| *Routing Layer - Sample Duration* | *Avg Depth* | *Avg Yield* | *Min Yield* | *Max Yield* | *Avg Power (mW)* |
|---|---|---|---|---|---|
| Standard - 30 | 1.9 | 0.53 | 0.14 | 0.88 | 5.4 |
| Standard - 150 | 1.5 | 0.50 | 0.05 | 0.92 | 1.4 |
| MINTRoute - 30 | 2.2 | 0.57 | 0.18 | 0.89 | 5.4 |
| MINTRoute - 150 | 2.1 | 0.57 | 0.08 | 0.93 | 1.4 |
| Low Power - 30 | 2.2 | 0.88 | 0.63 | 1.0 | 5.1 |
| Low Power - 150 | 2.1 | 0.84 | 0.32 | 0.98 | 3.6 |

TABLE II

ROUTING STATISTICS AND POWER CONSUMPTION PER NODE.

periods of 30s and 150s were run for 1 hour and 2.5 hours respectively, on a 15-node in-lab sensor network (Figure 11). Both queries collected routing tree parent, routing tree depth and node voltage. The power consumption of such queries thus reflected the underlying energy cost of queries in TASK, independent of any particular sensors used. For each configuration and query, we report in Table II average routing tree depth, average, minimum and maximum yield (per node) and average current consumption (measured on a single node). Figure 12 shows the yield averages for each node for the 30s query.

There are several immediate conclusions to be drawn from this data: first, MINTRoute provides better, more reliable routing than standard TinyOS routing (Figure 12 shows very high variance in the yield per node for standard routing, these results are consistent with [13]); the sample period has little effect on yield but a significant effect on power; low power listening provides significantly better yield (in particular, nodes 13 and 14 which are poorly connected to the rest of the network do much better), but at the expense of much higher power consumption for long sample periods (3.6mW vs 1.4mW at 150s); the standard routing and MINTRoute use the same amount of power (this is not surprising as power consumption is dominated by the 4s wakeup period and the cost of sending messages, which is equal in both cases).

With a 3V, 1000mAh battery (as used in the garden deployment), the 150s query would have a lifetime of 88

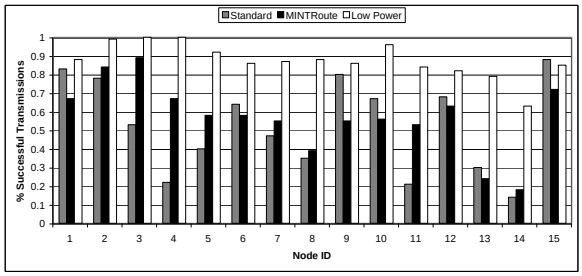


Fig. 12. Percentage of successful transmissions by sensor, 30s sample interval.

days with MINTRoute and only 34 days with low power listening. The reduced lifetime with the low power radio stack is due to at least three factors. First, nodes have a higher base power consumption when idle: the low power radio stack consumes 1mW when "idle" (checking every 100ms for a message), while duty cycling consumes 0.45mW when "off" (waiting for the next epoch). Second, sending messages costs more because they must have a 100ms preamble. Third, from examining our power consumption graphs, we see that nodes wake up from low-power mode to listen to messages which were not intended for them, thus spending a higher proportion of each epoch awake.

In conclusion, the current routing layers in TASK offer the choice of moderate reliability with low power consumption (MINTRoute) or better reliability but limited lifetime (MINTRoute with low power listening).

## V. LESSONS LEARNED

We first began development of TASK in April, 2002. Two years into the project, we continue to be surprised by the challenges of sensor network development, and to remark on the things we wish we had known when the project began. In this section, we briefly summarize some of those lessons.

**Be Pessimistic:** Failures in sensornets often occur in unexpected ways. For example, we spent months preparing for our initial deployment in the botanical garden, ensuring that sensors could be remotely reset, adding watchdog support, studying network reliability, and so on, only to have our first two weeks of deployment average a 50% uptime because the Windows-based laptop that was logging results repeatedly crashed.

Many of the lessons described below are instances of this "Murphy's Law" for sensornets. The implication is that all elements of a deployed sensornet system must be fault tolerant, must automatically detect failures and restart one another, and must be designed to assume unexpected faults will occur.

**Test at Scale:** A real fifty-node deployment in a target environment is quite different from five nodes on a table, or from a fifty node simulator. When we initially deployed TASK, we had tested our field tool with just a few nodes at a time. The first time we turned the field tool on in the botanical garden, all of the nodes hung and had to be reset. A subtle race condition in a flag in a message handler caused the software to stop sending messages. This bug was hard to reproduce with five sensors, but was inevitable with twenty-five.

**Build the Simplest Thing You Need:** Significant effort was expended to build unnecessary complex features such as TinyDB aggregation or precision time-synchronization. This lesson has important implications for many of the designers of sensor services like time synchronization [9], [10], communication scheduling [20], [21], and localization [22]. Long-term, low data rate applications such as TASK (and many other compelling sensornet applications) do not need high-precision time synchronization, fine-grained localization or a highly optimized TDMA scheme. Instead, these applications need highly reliable, low-overhead, and well-behaved implementations of such services. In our experience, many research implementations have subtle and hard to understand interactions with the rest of the OS that make them unsuitable for real-world use.

**Invest in Monitoring Infrastructure:** One of the explicit initial goals of TASK was to support system monitoring. The flexible, introspective query infrastructure provided by TinyDB, as well as the field tool, were a significant step in this direction. Over time this infrastructure only grew: we added multiple base stations to increase our ability to collect readings, wrote special parsing software to display the structure of messages transmitted over the radio, and added special logging code to write raw readings to motes' flash memory. Monitoring tools are the best approach to tackling the challenge of understanding and reproducing failures.

**Calibration is Hard:** Our initial assumption that the "calibrated" sensors we had purchased from vendors would produce accurate readings turned out to be completely false. Significant effort was required to calibrate each sensor individually. In our second round of garden deployments (which are currently underway), calibration has proved to be one of the most time consuming parts of the process, and is something that requires significant infrastructure (e.g., a controlled environment) and experience to do properly.

**Beware of Hidden Assumptions:** Software written for TinyOS is full of hidden assumptions, despite the best efforts of its authors to modularize the various pieces [23]. As a prime example of this, we found that on the 4 MHz `mica2dot` platform, the loss rates in TASK were significantly higher than on the 7 MHz `mica2` platform. The problem turned out to be our radio driver add-on that writes timestamps into messages. On the slower platform, the timestamp routine would cause the driver to miss inter-bit deadlines resulting in loss of radio synchronization and garbled packets. Unfortunately, TinyOS does not provide any built in mechanism to enforce runtime requirements on such low-level routines, and it is very difficult to judge how many instructions equate to a single radio bit time . While this lesson parallels the need to be pessimistic, it also calls attention to a class of invariants in sensornets that need to be checked and debugged. Over time, the lessons learned in identifying such assumptions should be reflected in the sensornet software engineering process and tools.

## VI. Related Work

Sensor networks have been deployed in 2002 and 2003 at Great Duck Island (GDI), off the coast of Maine, to monitor the habitat of the Leach's Storm Petrel [1], [24]. These sensor networks collected data on micro-climatic conditions on both the surface and in the Petrel's burrows. These deployments had a system architecture similar to that of TASK: the motes collected sensor readings and routed them to a central node, where the data was logged in a database. There was limited support for changing the data collection parameters. These deployments took the approach of building a simple, effective infrastructure for this specific environmental monitoring problem, as opposed to TASK which provides a general, reusable framework for sensor data collection.

The Extensible Sensing System (ESS) [25], [26] is being used to collect micro-climate data at the UC James Reserve in the San Jacinto Mountains of California, at more than one hundred locations. Like TASK, data is collected by sensor nodes and sent to a central server via a multi-hop wireless network. However, their are a number of significant differences in the system architecture: the sensor nodes in ESS are heterogeneous: some are motes like in TASK, while others are heavier duty (handheld-class, to support elaborate signal processing); the mote software is based on TinyDiffusion [27] rather than TinyDB; and the central server is based on a publish/subscribe model rather than on a submit-query/log-results-to-database model. At this point in time, ESS

is not a complete turn-key system, but is clearly more general than the GDI deployments.

The Zebranet [28] project uses sensor network nodes attached to zebra's to monitor their movements via GPS. More generally, Zebranet supports data collection from a sensor network composed of mobile nodes which are only occasionally in radio contact with each other or the base station. It must thus address a significantly different set of problems than TASK. No results from their recent deployment in the wild [29] are yet available.

## VII. Future Work and Conclusions

We are optimistic about the future of both TASK and the practical potential for sensornets as a whole. While our experiences building TASK have at times been frustrating, it seems that the TinyOS community is converging on reliable networking and power management services that will greatly improve future long term deployments.

At some point, if sensor networking technology is to have a significant impact outside of computer science, our community will need to shift some attention away from low-level micro-systems that work well in isolation, towards whole-system integration and configuration management issues. In particular, sensor calibration remains an open problem. We view TASK as an important first step in that direction, and the challenges we face in delivering TASK should provide useful context for other sensornet research.

## References


[1] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless Sensor Networks for Habitat Monitoring," in *Proceedings of the ACM International Workshop on Wireless Sensor Networks and Applications*, Sept. 2002.

[2] T. Dawson, "Fog in the california redwood forest: ecosystem inputs and use by plants," *Oecologia*, vol. 117, no. 4, pp. 476–485, 1998.

[3] T. Brooke and J. Burrell, "From ethnography to design in a vineyard," in *Proceeedings of the Design User Experiences (DUX) Conference*, June 2003, case Study.

[4] C. D. Patel, C. E. Bash, and A. Beitelmal, "Smart Cooling of Data Centers," U.S. Patent 6,574,104.

[5] C. Brown, "Casting a wider, deeper Net," EE Times, October 13, 2003, http://www.eetimes.com/at/news/OEG20031013S0025.

[6] "Future Store Initiative," http://www.future-store.org.

[7] S. Madden, W. Hong, J. M. Hellerstein, and M. Franklin, "TinyDB web page," http://telegraph.cs.berkeley.edu/tinydb.

[8] T. van Dam and K. Langendoen, "An adaptive energy-efficient MAC protocol for wireless sensor networks," in *Proceedings of SenSys*, 2003.

[9] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," in *OSDI*, 2002.

[10] M. Maroti, B. Kisy, G. Simon, and A. Ledeczi, "The flooding time synchronization protocol," Institute for Software Integrated Systems, Vanderbilt University, Tech. Rep. ISIS-04-501, 2004.

[11] http://www.xbow.com/Products/productsdetails.aspx?sid=65.

[12] "Jetty web server," http://jety.mortbay.org.

[13] A. Woo, T. Tong, and D. Culler, "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks," in *Proceedings of SenSys*, Nov. 2003.

[14] J. Hill and D. Culler, "Mica: a wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, November/December 2002.

[15] J. Zhao and R. Govindan, "Understanding Packet Delivery Performance In Dense Wireless Sensor Networks," in *Proceedings of SenSys*, Nov. 2003.

[16] D. Baldocchi, K. Wilson, and L. Gu, "How the environment, canopy structure and canopy physiological functioning influence carbon, water and energy fluxes of a temperate broad-leaved deciduous forestan assessment with the biophysical model CANOAK," *Tree Physiology*, no. 22, pp. 1065–1077, 2002.

[17] Intersema, "Ms5534a barometer module," Tech. Rep., October 2002, http://www.intersema.com/pro/module/file/da5534.pdf.

[18] T. A. O. Solutions, "Tsl2550 ambient light sensor," Tech. Rep., September 2002, http://www.taosinc.com/pdf/tsl2550-E39.pdf.

[19] Sensirion, "Sht11/15 relative humidity sensor," Tech. Rep., June 2002, http://www.sensirion.com/en/pdf/Datasheet_SHT1x_SHT7x_0206.pdf.

[20] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," in *IEEE Infocom*, 2002.

[21] B. Hohlt, L. Doherty, and E. Brewer, "Flexible power scheduling for sensor networks," in *IPSN*, Berkeley, CA, April 2004.

[22] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan, "The cricket location-support system," in *MOBICOM*, August 2000.

[23] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in tinyos," in *Proceeedings of NSDI*, March 2004.

[24] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler, "Lessons from a sensor network expedition," in *Proceeedings of EWSN*, January 2004.

[25] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Application driven systems research: Habitat monitoring with sensor networks," *Communications of the ACM*, June 2004.

[26] M. Hamilton, M. Allen, D. Estrin, J. Rottenberry, P. Rundel, M. Sri-vastava, and S. Soatto, "Extensible sensing system: An advanced network design for microclimate sensing," http://www.cens.ucla.edu, June 2003.

[27] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *MobiCOM*, Boston, MA, August 2000.

[28] 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 ASPLOS*, October 2002.

[29] "The ZebraNet Wildlife Tracker," http://www.ee.princeton.edu/~mrm/zebranet.html.