# SensorScope: Out-of-the-Box Environmental Monitoring

Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli
LCAV, I&C School, EPFL, Switzerland
{Guillermo.Barrenetxea, Francois.Ingelrest, Gunnar.Schaefer, Martin.Vetterli}@epfl.ch

Olivier Couach and Marc Parlange
EFLUM, ENAC School, EPFL, Switzerland
{Olivier.Couach, Marc.Parlange}@epfl.ch

## Abstract

*Environmental monitoring constitutes an important field of application for wireless sensor networks. Given the severity of potential climate changes, environmental impact on cities, and pollution, it is a domain where sensor networks can have great impact and as such, is getting more and more attention. Current data collection techniques are indeed rather limited and make use of very expensive sensing stations, leading to a lack of appropriate observations. In this paper, we present SensorScope, a collaborative project between environmental and network researchers, that aims at providing an efficient and inexpensive out-of-the-box environmental monitoring system, based on a wireless sensor network. We especially focus on data gathering and present the hardware and network architecture of SensorScope. We also describe a real-world deployment, which took place on a rock glacier in Switzerland, as well as the results we obtained.*

## 1 Introduction

### 1.1 Context

A Wireless Sensor Network (WSN) is a self-organized, multi-hop wireless network, composed of a large number of *sensor motes*, deployed over an area of interest. These motes are small embedded devices, able to gather various information about their environment, such as temperature, wind, humidity, or luminosity. They are constrained in many ways (*e.g.*, memory, processor), but energy is considered to be the scarcest resource, due to limited battery capacities. Moreover, as WSNs are often deployed in hostile and/or remote areas, replacing batteries may be infeasible.

Most of the time, WSNs operate in an *n-to-1* communication paradigm, in which collected data is forwarded to a base station (*sink*). The sink may then perform further computation on the data, or may in turn forward it via a longer-range/more reliable network connection (*e.g.*, wireline, GPRS).

WSNs may be divided into three categories:

1. **Time-driven:** Motes periodically forward gathered data to the sink (*e.g.*, pollution monitoring).
2. **Event-driven:** Motes forward an alert to the sink when a particular event occurs (*e.g.*, a forest fire).
3. **Query-driven:** Motes send gathered data only upon reception of a query from the sink (*e.g.*, storage room).

Typical uses of such networks include surveillance, habitat monitoring, and elderly care. We are, however, especially interested in environmental monitoring. Indeed, the natural environment is currently undergoing dramatic changes at a global scale, *i.e.*, global warming. Most of the time, environmental scientists cannot answer questions, such as "How much change is anticipated?" or "What are the main causes and consequences of such change?". The primary limitation in addressing these questions is a lack of appropriately dense spatial and temporal observations. Consequently, environmental researchers have difficulty to test and validate their models, which simulate future scenarios and make real-time predictions. An easy-to-deploy-and-configure pervasive WSN can greatly help in collecting the required data. This is our aim in the SensorScope project.

### 1.2 Contributions

In this paper, we present SensorScope, a WSN-based system for efficient environmental monitoring. SensorScope falls into the category of time-driven networks, as the stations intermittently transmit environmental data (*e.g.*, wind speed and direction, soil moisture) to a sink. This, in

978-0-7695-3157-1/08 $25.00 © 2008 IEEE
DOI 10.1109/IPSN.2008.28

turn, is able to relay to a database server, which makes all data publicly available in real-time on our *Google Maps*-based web interface and on Microsoft's *SensorMap* website[1]. The main objective of the project is to provide a low-cost and reliable WSN-based system for environmental monitoring to a wide community, to improve present data collection techniques with the latest technology.

By studying recent research results in WSNs, we have developed a communication stack that features, among other characteristics, a multi-hop data gathering protocol and a synchronized duty-cycling MAC layer that greatly helps in reducing the overall energy consumption. One of the key features of our solution is the very simple interface it presents to higher layers, abstracting all network details, and allowing for great ease when writing applications. This aspect is very important, since SensorScope aims at facilitating the adoption of WSNs as a common tool by a community with no expert knowledge in wireless networking.

As a case in point, during the summer 2007, we deployed several lightweight WSNs for typical environmental applications at the following locations in Switzerland:

1. **Morges**: Thanks to a collaboration with the Clim-Arbres project, a network was deployed on the border of the water stream Le Boiron de Morges. The Clim-Arbres project aims at renaturing this stream to improve its ecological quality, and was in need of appropriate environmental measurements.

2. **Le Génépi**: In collaboration with authorities, we deployed SensorScope in harsh conditions on a rock glacier on Le Génépi. This site is the source of frequent and dangerous mud streams, and the lack of measurements prevents climatic models from being elaborated. Our deployment allowed the gathering of the required measurements.

3. **Grand $S^t$ Bernard**: Again in collaboration with authorities, we deployed another network at the Grand $S^t$ Bernard pass. The goal was to create a very precise map of the evaporation in this place, thanks to soil water content and suction measurements, as current hydrological models have not represented reality well.

Therefore, the WSN concept, architecture, hardware, software, and web interface presented in this paper have had an immediate impact on real-world environmental monitoring applications. Here, we especially focus on how data gathering works and we hence describe both the hardware and the software architecture of SensorScope, the motivations that guided the design of our communication stack and how we implemented its prominent features. We provide results gathered during deployments about the network, the sensing stations, and, of course, the environment.

We also point out that the communication stack we have designed and describe in this paper, is freely available on our website[2], under an open-source license.

The remainder of this paper is organized as follows. In the next section, we present an overview of previous WSN deployments. We then present the architecture of SensorScope in Sec. 3, describing both the hardware we have designed and the communication stack we have developed. In Sec. 4, we give details about the key features of our network architecture, while in Sec. 5, we provide and discuss the results obtained on our indoor testbed as well as those of our aforementioned outdoor deployments. We finally conclude in Sec. 6, also pointing out future work we are considering.

## 2 Related Work

Wireless Sensor Networks have recently received quite some attention in the context of environmental monitoring [9, 13, 14, 16], as the highly distributed nature of relevant applications often renders wired deployments infeasible. Using inexpensive wireless sensor motes, it will eventually be possible to carry out measurement campaigns at unprecedented scales and resolutions, but because the technology is still relatively new, earlier experiments were generally small-scale and short-term.

Among the most recent papers, researchers at Berkeley reported the results obtained from their sensor network "Macroscope" [15], which was extensively used for the microclimate monitoring of a redwood tree. While this experiment provided quite interesting insights into deployment methodology and data analysis, it was a rather small-scale deployment. The nodes were placed in a tree from 15 to 70 m from the ground, and most sensor motes, especially the ones we use in SensorScope, are able to directly communicate over such distance.

Macroscope is built on top of TASK [1], a set of WSN software and tools, also designed at Berkeley. Authors of TASK state that the majority of substantial sensor network deployments have been driven and developed principally by network theorists and engineers, and that the adoption of WSNs as a generic tool requires the development of adequate and easy-to-use software. In SensorScope, our goal is actually to go one step further than TASK, and to provide not only a sensor network in a box, but an entire environmental monitoring system. This includes the hardware, such as the sensing stations, the software running on the motes, but also the server-side and database software, together with a convenient web interface.

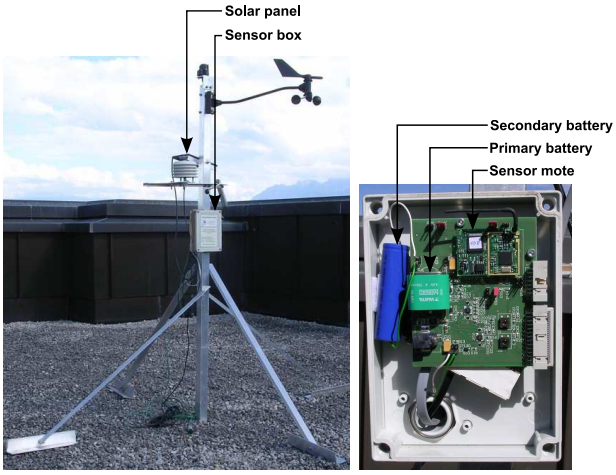
More recently, researchers at the Delft University deployed a large-scale sensor network in a potato field [6].

[1] http://atom.research.microsoft.com/sensormap/

[2] http://sensorscope.epfl.ch/network_code

The goal of the project was to improve the protection of potatoes against a fungal disease, and thus to precisely monitor the development of this disease. Unfortunately, the deployment went wrong and their work could not be finished, mainly because of time and money constraints. Their work, however, led them to report the lessons they learned, especially how much more difficult it is to set up a WSN in the real world rather than in a simulator.

Finally, we should point out that the SensorScope experiments presented in this paper are actually not the first ones. Previously, SensorScope was used for indoor monitoring, and was relying on bare motes, not fitted for outdoor, long-term operation [12]. The project is now much more mature, and supports deployments in harsh, isolated locations, as we shall show later on. The network architecture is also completely different.




(a) A sensing station. (b) A sensor box.

Figure 1: Design of a sensing station.

## 3 Project Overview

As mentioned above, a lack of correct and continuous observations is blocking the path to addressing crucial questions about our environment. Until now, there have been only limited field campaigns with in-situ spatial observations, and these campaigns have mostly been based on a small number of very expensive sensing stations, thus greatly restricting spatial coverage. Furthermore, these stations commonly use data loggers. This storing technique not only suffers from limited capacity, but also does not allow users to get immediate feedback from the system, since it requires manual downloading of the gathered data from each station. Thus, using a wireless sensor network is highly relevant to this area of research, as it allows for both real-time monitoring of imminent natural events (*e.g.*, storms, pollution) as well as long-term monitoring of persistent ones (*e.g.*, ice melting).

The SensorScope project aims at providing a new-generation environmental monitoring system, centered around a WSN, with a built-in capability to produce high-density temporal and spatial measurements. The system has to be effective out-of-the-box, with minimal requirements regarding network maintenance. Because the main objective is to replace the very expensive sensing stations that have been used until now, the baseline requirements that have been followed during its design are *low cost* and *full autonomy*, while maintaining sufficient accuracy for the intended application. In this section, we first describe the hardware architecture we designed, and then we provide an overview of our communication stack.

### 3.1 Hardware Design

When the project was started, there were no sensing stations with embedded sensor motes that could be used off-the-shelf. We, therefore, had to design and build suitable stations. The sensor mote platform we chose is a Shockfish TinyNode[3]. It is composed of a Texas Instruments MSP430 16-bit microcontroller, running at 8 MHz, and a Semtech XE1205 radio transceiver, operating in the 868 MHz band, with a transmission rate of 76 Kbps. The mote has 48 KB ROM, 10 KB RAM, and 512 KB flash memory. We opted for this platform mainly for its long communication range (up to 200 m outdoors) and its low power consumption [2].

The sensing station itself, depicted in Fig. 1a, is composed of a 4-legged aluminum skeleton on which a solar panel and the sensors are fixed. A station is 150 cm (60 in) high, and is thus both very stable, thanks to the 4 legs, and high enough to handle some snow build-up during winter. The sensor board is fixed inside a hermetic box, as illustrated in Fig. 1b, which is itself attached just above the legs. One can see the TinyNode mote on top of the board in this picture. The average price of such a station, including everything, is around € 900 ($ 1280).

#### 3.1.1 Power Source

In the spirit of Heliomote [11], we have designed a solar energy system to achieve sufficient autonomy during deployments. It is composed of three modules:

- **Solar panel:** A 162×140 mm MSX-01F polycrystalline module that provides a nominal power output of 1 W in direct sunlight, with an expected lifetime of around 20 years. We implemented a power control driver, following a strategy similar to that of Prometheus [5].

[3] http://www.tinynode.com

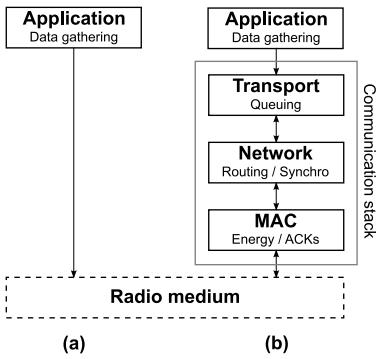



Figure 2: The first SensorScope software architecture (a) and the current one (b).

- **Primary battery:** A 150 mAh NiMH rechargeable battery (see Fig. 1b) is primarily used to power the stations. We chose a NiMH battery over a supercapacitor due to its superior capacity and its lower price.
- **Secondary battery:** A Li-Ion battery with a capacity of 2200 mAh at 3.7 V. It is the cylinder-shaped battery located on the left in Fig. 1b. This buffer is used as a backup source of energy during long periods of low solar radiation. It is charged via the primary buffer, thus undergoing fewer charging cycles.

This system, in conjunction with the power conserving algorithms implemented at the network level, theoretically makes the batteries' recharge cycle-count the only limiting factor for long-term deployments (see Sec. 5).

#### 3.1.2 Sensing Modalities

The stations can accommodate up to 7 different external sensors, some of them being able to measure multiple quantities. With our choice of sensors, the stations are capable of measuring 9 distinct environmental quantities: air temperature and humidity, surface temperature, incoming solar radiation, wind speed and direction, precipitation, soil water content, and soil water suction. Note that not all stations are equipped with all sensors, as SensorScope is perfectly able to cope with a heterogeneous set of sensors at each station. To ensure the quality of the measured values, all sensors are calibrated before deployment by comparing their readings to reference sensors over several days. The correlation coefficient obtained for the measured values is required to be higher than 0.98.

### 3.2 Network Design

The very first outdoor deployment of SensorScope occurred in July 2006 on the campus of EPFL, and it mainly aimed at validating the hardware design of the sensing stations. Accordingly, the software running on the motes was rather simple. The application, implemented in nesC for TinyOS [7], was not built on top of a real communication stack. This implied multiple limitations, especially in terms of range, reliability, and efficiency. Moreover, the application itself had to cope with network-related details, making it difficult, in case of a problem, to determine whether the network or the application was the culprit.

Gathering data in remote and difficult-to-access places (*e.g.*, Le Génépi deployment, described in Sec. 5) requires a robust system design, and we have found the assertion made in the TASK paper [1] to be true: simple and application-specific approaches provide the most robust solutions for real-world usage. Moreover, gluing existing components together takes a lot of time and effort for an in-depth understanding of their interactions. For these reasons, and to overcome the aforementioned limitations of existing systems, we chose to design and implement from scratch a communication stack for our stations with TinyOS. One of the main advantages of using such a stacking architecture is to completely separate the network management from the application, which just has to give the data to the stack and let it go to the sink by itself. Fig. 2 shows our stack, which is inspired by the well-known OSI model. The arrows indicate that currently, no data is forwarded to the application by the stack. The multi-hop mechanism is indeed automatically managed, and there is no need for the application to care about received packets. This may change in the future, for instance, if *in-network processing* is considered.

Our stack needs to store only 4 bytes of information per packet. We chose to put these into the payload, leaving 24 bytes for the application layer, out of the 28 available in TinyOS, as illustrated in Fig. 3. Another solution would have been to add our own header to the standard network header and to leave the TinyOS payload unchanged, but this would have implied to maintain the files after each new release of the radio drivers. Moreover, these files are radio-specific, and one would need to modify them each time a different mote/radio is used. By storing these bytes in the TinyOS payload, our stack is independent of the underlying radio drivers. In the following, we describe the different layers of our stack, starting from the highest one.

#### 3.2.1 Application Layer

This layer is quite simple and is only responsible for collecting the data that have to be sent to the sink. In SensorScope, it periodically queries both the sensors and the batteries, whose readings are used to monitor the energy level of the stations at the server. The values are then passed to the transport layer.

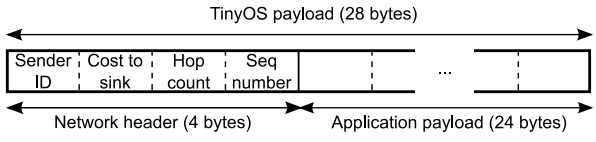



Figure 3: Format of a SensorScope packet.

#### 3.2.2 Transport Layer

This layer exposes a simple interface composed of two different commands for sending data. Each of them creates a different kind of packet:

- **Data packets:** They contain some data that must be routed towards the sink, examples of such data being the sensors' or the batteries' readings.
- **Control packets:** They are intended for a specific neighbor of the node, or to all of them, in case of a local broadcast, and they are thus not forwarded once received. Examples of such messages, which are detailed later on, are beacons or synchronization packets.

As we assume overall network traffic to be relatively low, this layer does not include any congestion avoidance mechanisms. It is responsible for creating packets out of the data received via the two aforementioned commands, and for storing them in the corresponding queue. Whenever one of the queues is not empty, this layer tries to send the next packet, by passing it to the network layer. Priority is currently given to control packets, *i.e.*, if there is both a data and a control packet waiting, then the latter is sent first. We chose this behavior because control packets are quite important for the network operation, and thus have higher timeliness requirements than data messages.

The transport layer is responsible for filling two fields in the network header (cf. Fig. 3). The first one is the hop count, which is set to 0 for newly created packets, and incremented each time a data packet is received. Note that this information is not mandatory, and is used only for statistical purposes. The second field is the sequence number, filled with an internal data or control message counter. These counters are incremented only when messages are correctly sent: if the sending fails for some reason (*e.g.*, no acknowledgment), the packet is resent with the same sequence number. This field is used for link quality evaluation, as explained later on in Sec. 4.1. Note that this layer is exactly the same for both the sink and the motes.

#### 3.2.3 Network Layer

This layer has to decide whether a packet should be routed towards the sink, based on its type (data or control), and if so, how this should be done. At the sink, this layer simply forwards data packets to the serial port, while control packets are passed to the MAC layer. At the motes, the network layer passes both kinds of packets to the MAC layer. Since control messages already have a recipient, no further action is required. For data packets, this layer first has to choose a next hop toward the sink. How this is done is protocol-specific and is detailed in the next section. Note that implementing a new routing protocol simply requires to write a new network layer, leaving the rest of the stack untouched.

The network layer is also responsible for filling the two remaining header fields, the sender identifier and the cost to the sink. How this information is obtained is, once again, protocol-specific, and detailed in the next section.

#### 3.2.4 MAC Layer

The MAC layer manages the radio itself, namely switching it on/off and sending/receiving messages. When a packet is received, it is immediately passed to the network layer. In case of a data message, an acknowledgment (ACK) is also sent back to the previous sender. Note that control packets are not acknowledged. In SensorScope, the MAC layer is also responsible for power management, as explained in Sec. 4.3.

When we prepared for our deployments, the radio drivers of the TinyNode were still lacking a *carrier sense*, and we could thus not add a busy-channel detection. Therefore, we decided to use a simple backoff mechanism, whose maximum delay is exponentially increased, each time a data packet is not acknowledged. Upon a successful transmission, the maximum delay reverts to the minimum value. Whenever sending fails because of a lost acknowledgment, the failure is signaled to the network layer with the appropriate flags. A busy-channel detection will be part of future releases of our code.

## 4 Networking

In this section, we describe the prominent features of the SensorScope communication stack that make the whole system auto-organized and energy-efficient, and how they are currently implemented. In the following, *broadcast* designates a local broadcast (*i.e.*, a packet sent to all neighbors), and not a network-wide one. The distance always designates the *hop-distance* to the sink, not the Euclidean one.

### 4.1 Neighborhood Management

For proper operation, nodes manage a *neighborhood table* in which they store the nodes they can hear from (literally their *neighbors*). A typical solution for nodes to acquire such knowledge is to let them regularly broadcast *beacon*

*messages*, containing their identifier; all receivers of such packets may then add the sender to their table. To reduce the network load, we chose to let nodes discover their neighborhood by overhearing neighbors' packets, in the spirit of MintRoute [17]. Only the sink has to send real beacons to initiate the process: upon receiving beacons, nodes at 1-hop distance start transmitting their data messages to the sink, letting nodes at 2-hop distance discover them, and so on. Each time a node updates its table, it also updates its cost to the sink. Note that we currently use the hop-distance to the sink as the cost metric. For instance, if a mote's best neighbors are at $x$ hops from the sink, then it assumes that its own distance, and thus its cost, is $x + 1$. Because neighborhood information is mainly needed for routing data to the sink, the table is managed by the network layer.

Due to the randomness of the radio channel, it is possible for a node to sometimes receive a message from a distant neighbor. In this situation, considering these nodes as neighbors may lead to routing problems, since messages would then have a low probability of being correctly received. To avoid this issue, it is required to provide the nodes with an estimation of the quality of service (QoS) of links, so that poor-quality neighbors may be considered separately, if applicable. To evaluate this QoS, the neighborhood table stores the sequence numbers of the last packets received, and the QoS is estimated by counting how many of them were not received: the quality then varies with the quantity of missing sequence numbers. When not enough messages were received from a given neighbor, we temporarily set its quality to 0. An alternative solution would have been to use the *received signal strength indicator* (RSSI), but we found this method not to be precise enough. The RSSI is indeed influenced by a lot of parameters (*e.g.*, antenna matching, location of nodes, ground effect), and the measured value for a given neighbor may greatly vary each time a packet is received.

Since the final goal is to route data messages to the sink, it is important for the estimated QoS of a neighbor to reflect its capacity to forward messages to the sink. A simple problematic situation may be, for instance, caused by a very good neighbor, in terms of QoS, with a poor capacity to route messages toward the sink. To avoid this, the sequence numbers of data messages are used to estimate the QoS. Indeed, when a neighbor is unable to successfully send a data message to a next hop, the message is resent with the same sequence number, thus decreasing the QoS of that neighbor. This mechanism ensures that the quality of a neighbor is based on how well it can be heard, as well as how good it is at "communicating" with the sink.

To account for *dead neighbors* (*e.g.*, hardware failure), the table needs to be cleaned from time to time. To do so, each entry has an associated timestamp, updated upon the reception of a packet, and a timer is used to regularly check how much time has elapsed since the last update of the corresponding entry. If that time is too long, then the neighbor is removed from the table. Regularly, a data packet is sent to the sink with the identifiers of the neighbors and their QoS, so that it is possible at the server to reconstruct the network topology. This greatly helps in identifying problems during deployments, due to the lack of good links between some stations.

## 4.2 Synchronization

Because of the randomness of radio connectivity, packets can be stuck at a node for some time, resulting in routing delays. This rules out that the server time-stamps the reports and implies that nodes have to put a timestamp in their reports to allow for meaningful interpretation of the sensed data. As our power management approach relies on duty-cycling (see next section), we opted for a global synchronization mechanism.

We have implemented this synchronization based on `SYNC_REQUEST`/`SYNC_REPLY` messages, the goal being to propagate the current time into the network from the sink, assuming it knows the current time. When a node wants to update its clock, it sends a request to a neighbor, *closer than itself* to the sink. If it knows the current time, that neighbor then broadcasts back a reply with this value. Upon reception of such a message, all nodes *further than the sender* from the sink update their clocks. Taking care of the distance ensures that the time always propagates away from the sink, while the sink simply puts it into its beacon messages. Broadcasting the replies helps in reducing the quantity of requests, since receivers postpone them, once their clock has been updated. Note that this synchronization mechanism is managed by the network layer, because it manages the distance information, but time-stamping is actually performed by the MAC layer since this eliminates delay errors, as observed by Ganeriwal *et al.* [3]. Another solution may have been to include the current time in the packet header, so that nodes could have updated their clock upon each packet reception. However, as a timestamp is stored on 4 bytes, this method would have decreased the available payload space to 20 bytes. Moreover, it is not needed to update clocks so frequently.

Because sensor motes are subject to *time drift*, clocks must be regularly updated. We decided to use two different update modes: a *high-frequency mode*, used when nodes do not have the current time (*e.g.*, after boot), and a *low-frequency mode* for later updates. The theoretical drift of the crystal used in TinyNodes is $\pm 20$ ppm (*i.e.*, around 1 s every 14 h), and Fig. 4 shows that the average experimental value is close to the theoretical one, with the amplitude getting quite large after some time. Based on these results, we consider that a period of 1 hour (average drift of 72 ms)

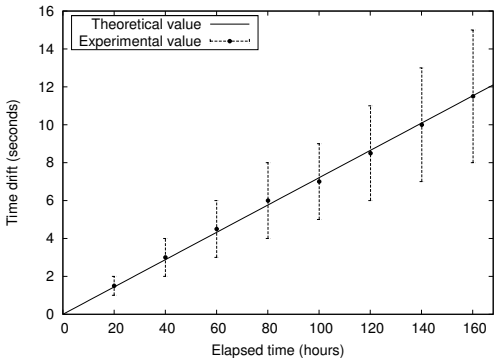



Figure 4: Theoretical and experimental time drift on a TinyNode (based on results with 7 motes).

for the low-frequency mode is sufficient. The particular choice of the high frequency is explained in the next subsection. With reasonable frequencies, this solution allows for synchronization of nodes within a few dozens of milliseconds. Although some high-accuracy solutions exist, such as FTSP [8], our approach is simple and provides sufficient precision for both time-stamping and duty-cycling. Moreover, high-accuracy solutions compensate time drift with linear regression, while the drift, however, can vary depending on weather conditions, making it quite difficult to completely avoid synchronization errors. From our experience, if the application allows it, it is better to live with a slight drift rather than trying to eliminate it.

We first decided to regularly send the real time from the server to the remote sink, but we found this method to be problematic: when using GPRS to forward data from the sink (*e.g.*, the Génépi deployment), it is difficult to send data from the server to the sink. We thus chose to use the local time of the sink as the *network time*, and to translate timestamps at the server. To achieve this, the sink regularly sends a message with its local time to the server, which in turn can compute the offset between the network time and the real time. To account for accidental reboots of the sink, it first tries to synchronize with other nodes, by broadcasting requests, using the high-frequency mode. In case of no reply (*i.e.*, the network has just been started), it starts using its own local time as the network time, which then propagates.

### 4.3 Power Management

Power management is essential for long-term operation, and, although our solar energy system is quite efficient, the mote's radio is a big energy consumer: keeping it on all the time may lead to a negative energy balance (*i.e.*, with regards to the incoming solar power). A quick look at the TinyNode's data sheet shows that the energy consumption is equal to 2 mA when the radio is off, while it is equal to 16 mA when the radio is on for reception. This means that turning off the radio as frequently as possible, rather than listening constantly, reduces energy consumption by a factor of approximately 8.

For energy-efficiency, nodes thus have to organize themselves into two-state communication cycles: an *active state*, where the radio is on for sending/receiving messages, and an *idle state*, where the radio is off. Achieving good energy savings, of course, requires the idle state to be as long as possible. Two major mechanisms exist:

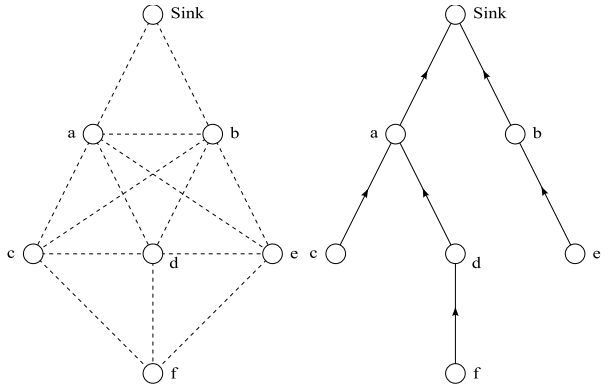
1. **Low-power listening** (LPL). This solution is asynchronous, meaning that nodes do not have to wake up at the same time to communicate. To achieve this, a *preamble* (*i.e.*, a specific pattern of bits) is sent before the packet itself. If its length is longer than the idle state, all neighbors are ensured to detect it during their upcoming active state, and to wait for the incoming packet. B-MAC [10] is a well-known MAC layer that uses this mechanism.

2. **Duty cycling**. In contrast, this solution requires all nodes to synchronously switch their radio on. Because they are all active at the same time, there is no need for preambles and packets can be sent as usual, resulting in slightly better savings upon transmissions. TASK [1] makes use of duty cycling to conserve energy.

Although we were almost forced to opt for the duty cycling method because of the lack of a carrier sense of our radio drivers, we found this solution to be generally better than LPL. LPL indeed requires the preamble to be longer than the idle state, and since good energy savings require this state to be long, transmissions can themselves get very long, resulting in congestions when the traffic level is not low enough. It has also been shown that LPL may actually decrease a mote's lifetime compared to duty cycling because of a slightly higher energy consumption [1].

Moreover, waking up nodes at the same time is easily done, thanks to the synchronization mechanism previously described, which is precise enough for this purpose. To take care of the startup, when nodes do not have the network time, they keep their radio on until being synchronized, and the high frequency mentioned in the previous subsection must thus be chosen carefully. To ensure that a request will be received by a neighbor during its next communication cycle, the delay used in this mode simply has to be smaller than the length of the active state. To account for a slight time drift, a node first waits for a few dozen milliseconds, without sending messages at the beginning of its active state, to ensure that its neighbors are indeed awake.

Note that this energy-saving mechanism is completely managed by the MAC layer, and is transparent to the other




(a) The network topology. (b) A possible backbone.

Figure 5: Using a backbone for data gathering greatly reduces the possibilities to reach the sink.

ones. When a message has to be sent while the node is idle, then the message is kept and sent only during the next active state. Since upper layers wait for the *sendDone* signal (TinyOS is based on split-phase operations), the actual waiting time does not matter.

### 4.4 Routing

To route data messages to the sink, a possible solution is to maintain a backbone, generally a tree rooted at the sink itself, such as the one illustrated in Fig. 5b. This implies a maintenance cost to detect broken links, but also quite an effort of organization to balance the network load between the different possible routes. Indeed, without any further provision, one could imagine a situation where all 2-hop nodes would use the same 1-hop node as their next hop. This node would then become a bottleneck, and would spend most of its energy forwarding not only the messages of 2-hop nodes, but also the messages of 3-hop nodes, 4-hop nodes, and so forth. This situation may, for instance, happen when motes are linked to their best parent (with regards to the considered metric), such as in MintRoute [17].

To avoid such problems and the maintenance of a routing structure, we decided to let nodes choose their next hop at random each time a packet has to be sent, resulting in an alleviated form of *opportunistic routing*. In WSNs, it is indeed not really important to take care of which route is used to reach the sink, provided that it eventually gets all data messages. Fig. 5 clearly illustrates that philosophy: using a backbone, such as the one in Fig. 5b, constrains node $f$ to use $d$ as the next hop for all its messages, while there is no reason to not use either node $c$ or node $e$. Moreover, node $a$ has to support 3 nodes ($c$, $d$ and $f$) while node $b$ supports only $e$, resulting in poor load balancing. Using a different next hop each time results in automatic load balancing, and each node is used in the best possible way, based on the underlying topology.

While always selecting a next hop at random inherently provides good load balancing between all possible neighbors, it is, however, of interest to favor good neighbors. To achieve this, we defined two thresholds: all neighbors with a QoS above the first one are considered as *high-quality neighbors*, while other ones above the second threshold are *low-quality neighbors*. When a message needs to be forwarded, one of the high-quality neighbors is chosen at random. If none exists, the algorithm randomly picks a low-quality one. Neighbors under the low-quality threshold are not considered at all.

Table 1: System parameters used during deployments.

| Layer | Parameter | Value |
|---|---|---|
| Application | Sampling time | 120 sec |
| Network | High-quality links | $\geq 90\,\%$ |
| | Low-quality links | $\geq 70\,\%$ |
| | Neighbor timeout | 480 sec |
| | High sync freq | 5 sec |
| | Low sync freq | 1 h ($\pm$ 72 ms drift) |
| Mac | Active state | 12 sec |
| | Idle state | 108 sec |

## 5 Experimental Results

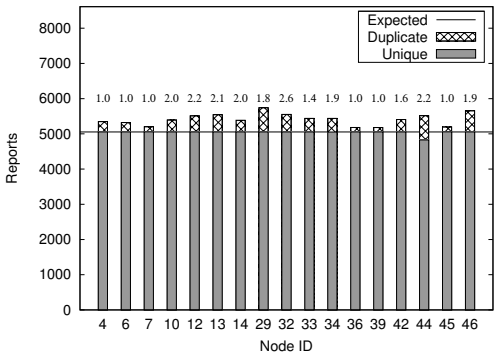
In this section, we provide some of the experimental results, we gathered during our indoor and outdoor experiments. Table 1 shows the different parameters, we used for both of these experiments.

### 5.1 Indoor Experiments

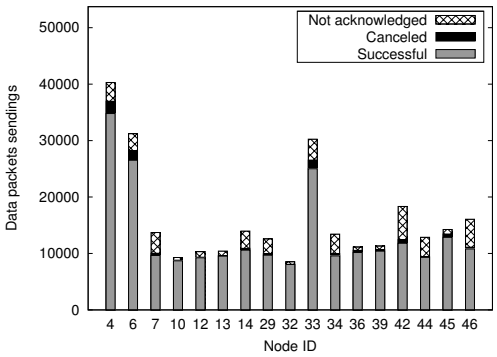
We first tested the network code on our testbed, which is composed of TinyNode motes, deployed in our office building on different floors. Using the same motes in both the testbed and the sensing stations is extremely important because different drivers, especially the radio drivers, may have different behaviors, and while the code could work just fine on some motes, it could have problems on other ones.

Of course, these motes are not wired to any external sensors since the testbed is used only to test the network code. All of them are plugged into AC power, allowing us to disregard any problems linked to energy management. Moreover, all motes are equipped with a Digi Connect ME module[4] which makes it possible to access and program the motes over an Ethernet connection. Each Digi module is

[4] `http://www.digi.com`




(a) Data gathering reliability.




(b) Load distribution of the network.

Figure 7: Results of the testbed experiment over one full week of operation.

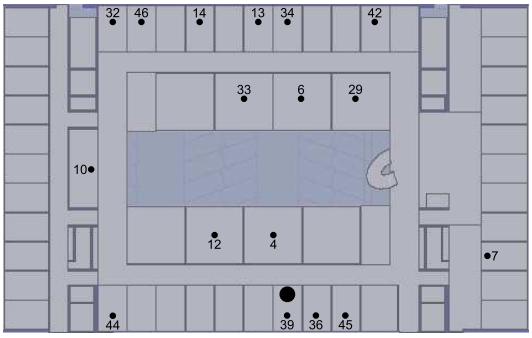



Figure 6: The map of our testbed.

indeed assigned an IP address which, in combination with the appropriate PC-side drivers, allows for transparent PC–mote serial communication. Such modules are very important to allow for quick testing, and having to flash the motes manually, by using a real serial port, would only be a waste of time.

For the test run presented here, we deployed the code on 17 motes of our testbed and let it run for one full week. Fig. 6 provides a map of them, node 29 being one floor below the other ones. The sink is symbolized by the big circle at the bottom of the map. The approximate dimensions of the building are $62 \times 40$ m. One should note that the center part of the building is empty, letting nodes communicate through it. At that point, we did not care about measuring energy consumption since the external sensors can consume quite a bit, and they are not present in the testbed.

Fig. 7a shows how many reports were received from the motes, the numbers above the bars giving the average hop count for the whole run. Thanks to the MAC-layer acknowledgment mechanism, we were able to receive all reports, except from node 44. It seems that this one got disconnected for some time, and its data message queue overflowed, resulting in a loss of more or less 200 reports. We could not really increase the maximum number of hops because of the aforementioned long range of the TinyNodes, but after all we chose them mainly for that good property. The furthest nodes were 32 and 44, their distance varying between 2 and 3 hops.

Duplicate packets, which appear upon the loss of acknowledgments, were kept at an acceptable level during the run, the average percentage of them being around 6.5%. Duplicates cannot be easily filtered out of the network because a random next hop is chosen each time, being a retransmission or not. So when a node receives a packet, chances are that if it is a duplicate, this node will not be able to determine it and thus to drop it. A possible improvement could be of course to filter out duplicates at least when twice the same next hop is chosen.

Fig. 7b shows the total quantity of sent data packets for each node, including all those packets forwarded because of the multi-hopping mechanism. All kinds of data packets are included in this figure (*e.g.*, reports, network statistics). Not surprisingly, nodes 4, 6, and 33 were the ones with the highest number of sent data packets. Indeed, due to their central location in the building, they were mainly used as next hops by the nodes in the upper part of the map. Node 12 also has a central location, but for some reason, it was not able to communicate directly with the sink. Node 29, being on a lower floor, was most of the time at 2 hops from the sink and was thus not used by upper motes.

Most of the time, canceled sendings occur when a data packet is received and an ACK has to be sent with high priority. In this case, the current backoff, if any, is canceled by the MAC layer to immediately send the ACK. A good example of such a situation is node 33: because it was highly used as a relay, a lot of its sendings were canceled. In contrast, node 36 has a low rate of cancellation because it was not really used as a next hop.

(a) Global view. (b) Local view.

Figure 8: The map of the Génépi deployment.

### 5.2 Outdoor Deployments

Based on these results, we performed a small deployment (10 stations) on EPFL campus to test the code on the actual stations. Of course, the network did not work at first because of some bad interactions between the code and the various drivers used on the stations (*e.g.*, solar panel, external sensors), but we could quickly resolve these issues. We decided to keep this network up and running, as a test deployment for future versions of the code.

During the last 15 months, we have run 6 outdoor deployments, ranging in size from 6 to 97 stations, from the EPFL campus to high mountain. Due to space limitations, we solely focus on our most important deployment, which occurred on a rock glacier located at 2 500 m on Le Génépi, in Switzerland. This site was chosen because it is always the source of dangerous mud streams during intense rain, and several people were killed because of them in the last decade. The authorities in charge did not have any measures of rain at that site, and asked us to deploy SensorScope there, the final goal being to correlate rain measurements with wind and temperature, based on the shape of the landscape. They gave us all the needed technical help for this deployment, including a helicopter and a container to sleep in at night. The sensing stations were deployed during the last days of August 2007 and taken down again two months later, in October 2007.

The 16 stations were deployed on a 500×500 m area (see Fig. 8). Special care was taken to put them at good locations, in order to retrieve meaningful measurements for environmental monitoring and modeling. For instance, station 20 was specifically put at the dislocation border of the glacier, and station 11 in the soil slope. To transmit the packets to the server, the sink, placed close to station 3, was equipped with a GPRS module. This was actually the first time we used such a module for a real-world deployment, and although the connectivity was quite poor on the site, it was sufficient for the deployment to be successful.

We believe that visual feedback is important to assess the development of a potentially critical situation (*e.g.*, avalanches, rock slides) or to better interpret gathered data (*e.g.*, presence of snow). Thus, in collaboration with an industrial partner[5], we have developed an autonomous, smart camera. The first version of this project resulted in a standalone camera that was installed on the glacier and transmitted a 640×480 image of the deployment every 30 min. Since the traffic generated by the camera is higher than the environmental data as a whole, the camera uses an independent GPRS connection. We do not give much details about the project in this paper because of lack of space, but transmitted images may be viewed on our website[6].

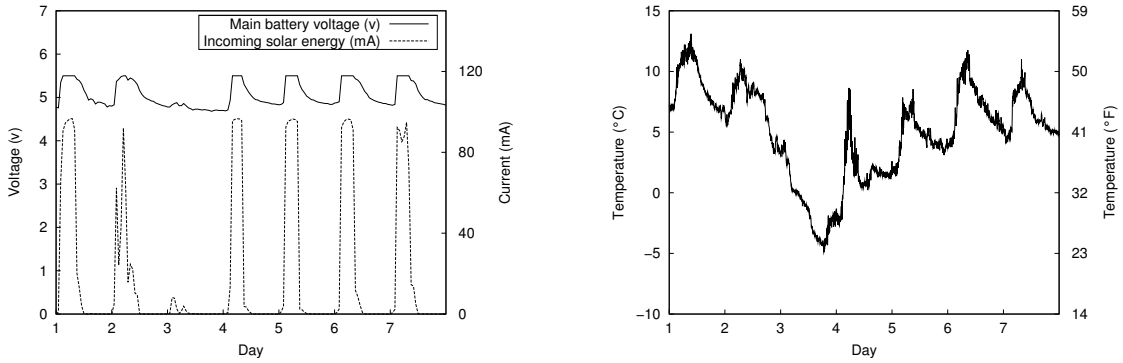
This deployment was a very good opportunity for us to thoroughly test the autonomy of the stations in real and harsh conditions. Fig. 9 shows the variation of energy of station 15 for one whole week, the associated incoming solar power and the correlation with the observed air temperature at that station. The observation started at 06h00 in the morning of the 16th of September 2007, and during that period the sunrise was around 06h00 and the sunset around 21h00. These results are more realistic than just considering the mote's consumption, since here we consider external sensors which can consume quite some energy.

One can clearly see that the main battery slowly depletes during periods of low solar radiation, obviously at night, and starts charging upon the sunrise until being fully charged during full daylight. On day 3, the weather was very cloudy, resulting in a brutal drop of the temperature, but the incoming solar power was still high enough for the battery to charge sufficiently. During the whole week, the secondary battery was actually not used at all, and would have powered the system in case of a failure of the primary one. Overall, we are satisfied with the energy system, since all batteries were always fully charged during the whole deployment, and even if some other hardware failures occurred, we did not have to worry about the energy level.

Fig. 10 provides the sensing reports gathered during one full month period, starting from the 10th of September 2007. We used the same set of parameters as for the indoor experiments, and we were able to collect all the reports from 10 stations. Some hardware failures occurred with the other ones, especially station 7, and we had to go back to the site to repair them. Because of the importance of this deployment, we absolutely wanted it to be successful and we thus used a very conservative approach that resulted, in conjunction with the clear outdoor environment, in having many stations at more or less one hop from the sink. Because of this, there are less duplicates than during the testbed run, but also because some interferences we may have in our building do not exist on top of a mountain.

[5] http://www.quividi.com

[6] http://sensorscope.epfl.ch/vidicam




(a) Available energy.

(b) Observed air temperature.

Figure 9: One week of data from the solar energy system of a weather station. The first day starts at 06h00 in the morning.

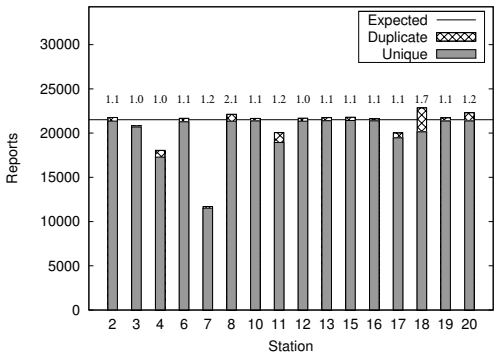
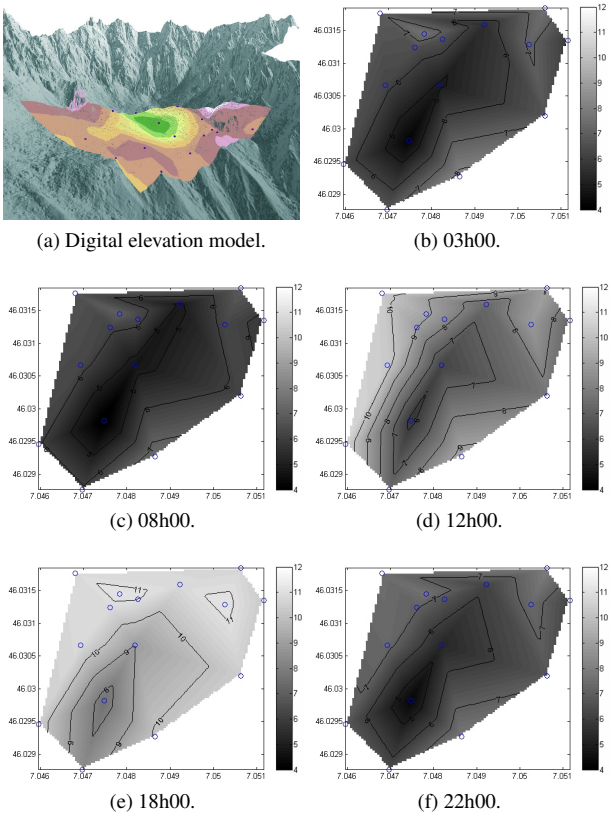



Figure 10: Reports gathered during the Génépi deployment and the average distance of the stations from the sink.

Fig. 11a shows the location of the stations on the digital elevation model of the rock glacier. One can see the valley in the center of the picture, which is where the permafrost is the thickest (around 10 to 15 m of ice under the rocks). This is also where the Durnand river is rooted, which is the source of the dangerous mud streams. The other maps of Fig. 11 show the spatial distribution of the air temperature during the 2nd of October 2007. During that day, there was perfect sunny weather, with a light wind from the south. Each value is the average of the measurements of one hour.

These results show that even during sunny days, the temperature is always very low along the valley of the rock glacier. While the variation of temperature is around 5°C on the border of the site, the maximal variation of the valley is only around 2°C. This is interesting because the corresponding stations are placed along the same axis and face the same sun exposure, they should thus observe the same temperature. This difference is actually caused by the thick layer of ice under the granite rocks, located along the valley.



(a) Digital elevation model.

(b) 03h00.

(c) 08h00.

(d) 12h00.

(e) 18h00.

(f) 22h00.

Figure 11: Digital elevation model (0.5×0.5 m resolution) and spatial air temperature distribution over the Génépi rock glacier along the 2nd of October at 03h00, 08h00, 12h00, 18h00 and 22h00 (local time).

Thus, the temperature is always kept low at that place, even with exposure to the sun during the day. Thanks to SensorScope, we were able to identify this microclimate on the Génépi, which plays an important role in the model, used to predict the mud streams.

## 6 Conclusion

Throughout the various deployments, SensorScope matured into a key project, merging cutting-edge wireless sensor technology (networking, sensing, hardware, software) with leading environmental monitoring (modeling, prediction, risk assessment). In particular, the Génépi deployment has been a thrilling scientific adventure, which resulted in the gathering of a unique set of meteorological data. This allowed us to model a particular microclimate, which can be used in flood monitoring and prediction, potentially reducing a well-known, but poorly understood, environmental hazard. We strongly believe in the potential of SensorScope for such risk prevention.

This deployment also revealed how remote management is crucial in such harsh conditions. Dynamic reconfiguration of network and motes is our next main objective, and support for a system, such as Deluge [4], is of high interest. From the network management point of view, we also plan to implement measures to cope with asymmetric links, which result in transmission failures and an overly high radio usage. Finally, due to the difficult measurement conditions, the measured data is of variable quality. Thus, signal processing techniques for better calibration, detection of outliers, denoising, and interpolation will be developed.

## 7 Acknowledgments

This work is partially financed by the Swiss NCCR MICS, the European FP6 project WASP, and Microsoft Research.

## References

[1] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden. TASK: Sensor network in a box. In *Proceedings of the IEEE European Workshop on Wireless Sensor Networks and Applications (EWSN)*, Jan. 2005.

[2] H. Dubois-Ferrière, R. Meier, L. Fabre, and P. Metrailler. Tinynode: A comprehensive platform for wireless sensor network applications. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2006.

[3] S. Ganeriwal, R. Kumar, and M. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2003.

[4] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at a scale. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.

[5] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor network. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2005.

[6] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the IEEE international Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2006.

[7] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, K. Whitehouse, J. Hill, M. Welsh, E. Brewer, D. Culler, and A. Woo. *Ambient Intelligence*, chapter TinyOS: An Operating System for Sensor Networks. Springer, 2005.

[8] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.

[9] K. Martinez, J. Hart, and R. Ong. Environmental sensor networks. *IEEE Computer*, 37:50–56, 2004.

[10] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2004.

[11] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava. Design considerations for solar energy harvesting wireless emebedded systems. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2005.

[12] T. Schmid, H. Dubois-Ferrière, and M. Vetterli. Sensorscope: Experiences with a wireless building monitoring sensor network. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN)*, June 2005.

[13] P. Sikka, P. Corke, P. Valencia, C. Crossman, D. Swain, and G. Bishop-Hurley. Wireless adhoc sensor and actuator networks on the farm. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, Apr. 2006.

[14] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the IEEE European Workshop on Wireless Sensor Networks and Applications (EWSN)*, Jan. 2004.

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

[16] G. Werner-Allen, J. Johnson, M. Ruiz, M. Welsh, and J. Lees. Monitoring volcanic eruptions with a wireless sensor network. In *Proceedings of the IEEE European Workshop on Wireless Sensor Networks and Applications (EWSN)*, Jan. 2005.

[17] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the ACM International Conference on Embedded Networked Sensor Systems (SenSys)*, Nov. 2003.