On Meeting Lifetime Goals and Providing Constant Application Quality

ANDREAS LACHENMANN
European Microsoft Innovation Center
KLAUS HERRMANN and KURT ROTHERMEL
Universität Stuttgart
and
PEDRO JOSÉ MARRÓN
Universität Bonn and Fraunhofer IAIS

Most work in sensor networks tries to maximize network lifetime. However, for many applications the required lifetime is known in advance. Therefore, application quality should rather be maximized for that given time. *Levels*, the approach presented in this article, is a programming abstraction for energy-aware sensor network applications that helps to meet such a user-defined lifetime goal by deactivating optional functionality.

With this programming abstraction, the application developer defines so-called *energy levels*. Functionality in energy levels is deactivated if the required lifetime cannot be met otherwise. The runtime system uses data about the energy consumption of different levels to compute an optimal level assignment that maximizes each node's quality for the time remaining.

As described in this paper, *Levels* includes a completely distributed coordination algorithm that balances energy level assignments and keeps the application quality of the network roughly constant over time. In this approach, each node computes its schedule based on those of its neighbors.

As the evaluation shows, applications using *Levels* can accurately meet given lifetime goals with only small fluctuations in application quality. In addition, the runtime overhead both for computation and for communication is negligible.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.8 [**Operating Systems**]: Performance; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms: Design, Performance, Measurement

Additional Key Words and Phrases: Wireless sensor network, energy, lifetime goal, programming abstraction, coordination

This article extends work published in the Proceedings of ACM SenSys 2007. The research has been done at Universität Stuttgart. It has been supported, in part, by Landesstiftung Baden-Württemberg.

Author's address: European Microsoft Innovation Center GmbH, Ritterstr. 23, 52072 Aachen, Germany; e-mail: andreas.lachenmann@microsoft.com

1. INTRODUCTION

Energy limitations are paramount in sensor networks. Traditionally, most research has tried to minimize energy consumption and to maximize network lifetime. Although this is often useful, for many sensor network applications the required lifetime is known in advance. Therefore, instead of *maximizing lifetime*, it is more important that the nodes stay alive for a *user-defined lifetime* and that during this time the application provides the *best quality* possible subject to the energy constraints present.

For example, in long-term structural health monitoring of bridges [Kim et al. 2007; Marrón et al. 2005] the batteries of nodes can only be replaced every few years during regular inspections [Marrón et al. 2005]. As the interval of these inspections is known beforehand, it corresponds to the lifetime goal of the network. Likewise, in many environmental monitoring applications such as observing volcanoes [Werner-Allen et al. 2006], the microclimate of trees [Tolle et al. 2005], or wildlife [Liu et al. 2004] the required lifetime is known in advance because the scientists involved know the desired duration of their experiment. If some nodes fail early, either network connectivity cannot be preserved or the resolution of the sensor data is reduced.

In these – and other – applications it is possible to identify parts which are more energy-intensive than others and which are not actually needed to provide some basic functionality. For example, such a task can be writing a backup copy of the sensor data to flash memory in addition to transmitting it to the base station. If the network topology is sparse and powering the sensor interface board is energy-expensive, some nodes could even stop sensing in order to preserve network connectivity.

However, developing adaptive applications is difficult: Such an application has to constantly monitor energy consumption and the node's lifetime. Furthermore, it has to include algorithms that adapt the application's functionality to reach the given lifetime while dealing with possibly inaccurate energy measurements on inexpensive hardware. All of this makes writing such applications hard and increases development overhead significantly. Therefore, we have created *Levels*, an abstraction for energy-aware programming of sensor networks that allows the developer to explicitly single out optional functionality [Lachenmann et al. 2007]. Using our system, the developers do not have to manually deal with low-level energy issues (e.g., estimating the remaining energy and current energy consumption) or with the adaptation algorithms themselves. Therefore, the effort for developing adaptive applications that meet a given lifetime goal is significantly reduced.

With our approach, developers can specify so-called energy levels in an application which differ in their energy consumption and the functionality they offer. The code within each such level is associated with the amount of energy it consumes. At runtime *Levels* monitors the remaining battery capacity and the energy consumed by each level. It then selects an energy level that allows the application to achieve its target lifetime, if necessary, with restricted functionality. Compared to manual implementations of such adjustments this programming abstraction and its corresponding runtime system save much development effort.

Our approach is based on measuring the energy consumption of individual energy levels using an energy profiler with accurate simulation models [Titzer et al. 2005;

Landsiedel et al. 2005]. At runtime each node tries to maximize the utility of the energy levels while achieving its lifetime goal. As we show in the evaluation, the abstraction of energy levels is useful in real-world applications and *Levels* is able to help meeting lifetime goals while providing good application quality.

However, if all nodes perform this optimization individually and if their energy load is roughly equal, they will probably switch to similar energy levels at the same time. Then the overall application quality of the network – i.e., the average utility of all nodes – will be either excellent or poor but nothing in between. To better distribute energy level assignments over time, we propose two different approaches. These approaches increase the probability that, for example, there are always some nodes gathering data even if some have deactivated their sensors to meet their lifetime goal. Both of them continue to use the local optimization result that indicates how long each energy level should be active. The first approach simply selects a random energy level from this result, which will be activated immediately; the other energy levels that are part of the result will still be selected later. If the nodes have similar optimization results, this will lead to a uniform assignment of energy levels. The second approach minimizes variations even more by coordinating the energy level assignments with neighboring nodes in a completely distributed way.

Apart from balancing energy level assignments, these approaches can also be applied to activate and deactivate nodes if some of them are redundant. With such additional nodes a longer network lifetime can be achieved because it is sufficient if each node is active for a fraction of the overall network lifetime. Deploying redundant nodes is possible in many applications. It is often easier than replacing batteries or deploying additional nodes later if the nodes are placed in inaccessible locations. Usually, the effects on application quality when activating or deactivating nodes are even greater than when switching energy levels.

Our solution has several benefits. First, the developer does not have to deal with the low-level issues of energy consumption, which simplifies the development of energy-aware applications. Second, our solution helps to ensure that a given lifetime is reached and that good application quality is offered. Third, with its distributed coordination our approach avoids fluctuations in application quality. Fourth, the overhead for the developer is small and we provide a powerful programming abstraction that allows for modular application development. Finally, the runtime overhead of our system is negligible.

The rest of this paper is structured as follows. Section 2 describes the energy levels abstraction and the local optimization done on each sensor node. Section 3 then explains our distributed approach that balances energy level assignments. In Section 4 we describe limitations of our abstractions and algorithms. After that, Section 5 presents evaluation results for both the local optimization and the distributed approach. Finally, Section 6 gives an overview of related work, and Section 7 concludes this paper.

2. MEETING LIFETIME GOALS WITH ENERGY LEVELS

In this section, we present the abstraction of an energy level. First, we describe relevant design characteristics and the abstraction itself. Then we detail the corre-

sponding runtime system including its battery model, its mechanisms to attribute energy consumption to an energy level, and the local optimization of energy levels.

2.1 System Design

2.1.1 *Sensor Network Properties.* Several properties of wireless sensor networks aid our approach of measuring energy consumption and switching between energy levels at runtime.

—There is usually just a single application running on each sensor node. Therefore, the expected lifetime of a node depends only on one application that can be controlled more easily than a multitasking system.

—Sensor networks typically exhibit some periodic behavior. For example, sensor readings are sampled periodically at user-defined time intervals. If the sensor network application reacts to external events, these events often repeat for sufficiently large periods. Thus, it is possible to estimate future energy usage based on past consumption.

—Because sensor nodes only have limited output capabilities and are often deployed in inaccessible locations, simulation has become an integral part of the development process [Titzer et al. 2005]. In addition, simulators are often equipped with detailed energy models [Landsiedel et al. 2005; Shnayder et al. 2004]. Therefore, we can use simulation to get information about the energy consumption of a piece of software.

—Most sensor nodes available today are equipped with voltage sensors. Since the voltage provided by a battery depends on its remaining capacity, we can use the voltage data to estimate the residual energy.

—As the nodes are strictly energy-constrained, in the domain of sensor networks software developers are more concerned about energy consumption and node lifetime than developers in other areas. Therefore, we expect that most developers are willing to invest some effort for specifying energy levels and measuring energy consumption.

2.1.2 *System Overview.* Our programming abstraction of "energy levels" allows to specify optional code blocks. At runtime, the system decides which energy levels are active, i.e., which code blocks should be executed. This abstraction is described in more detail in Section 2.2.

Basically, our system is similar to well-known Model Predictive Control (MPC) schemes [Camacho and Bordons 2004]: First, we build a model that helps to predict energy consumption by profiling the energy consumed by optional code (see Section 2.3). This model can be used to compute the energy consumed by each level at runtime. Second, it is complemented by a battery model that maps voltage readings to the remaining energy usable by the sensor node (see Section 2.4.1). Third, using the information from energy profiling, *Levels* keeps track of how much energy is consumed by each energy level at runtime (see Section 2.4.2). This part of *Levels* considers both energy that is consumed just once when executing a code block (e.g., to store some data in flash memory) and changes in the rate of continuously consumed energy (e.g., by enabling a sensor). Finally, together with the battery model, this data allows to compute the expected node lifetime in each energy level.

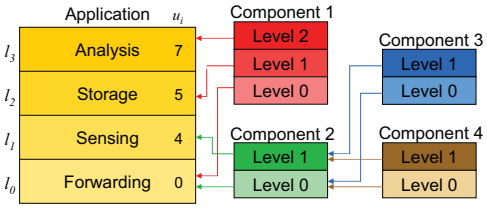


Fig. 1. Combining energy levels

This information is then used to optimize the energy level for the remaining lifetime considering the given energy constraints (see Section 2.4.3). Like other MPC algorithms, our system considers just the result for the current time interval and later recomputes the remaining level assignments to better reflect the new situation.

2.2 Energy Levels

An energy level includes all statements that should be active at the same time. It can be deactivated at runtime and, therefore, is optional for providing some basic functionality of an application. If an energy level is deactivated, however, the functionality of the application may be degraded.

If an energy level is active, all levels below it are also active so that higher levels can rely on the functionality of lower ones. Nevertheless, having such an order of energy levels does not mean that the functionality of an application has to increase monotonically with higher levels. Although the levels below the current one are always active, the application developers can still define appropriate conditions to specify that the code within a low level should not be executed if a higher level has been activated. This way they can write an application that, for example, transmits sensor readings in a low energy level to the base station and stores them just locally (without forwarding them) in a higher level.

Levels assumes that higher levels lead to an increase in energy consumption. Otherwise, energy levels should be merged because they are ill-defined. Such a situation could be easily detected during the development phase. Furthermore, the runtime system assumes that higher energy levels provide better application quality. To represent this quality, each energy level l_i is associated with a utility value u_i: The application developer can define this utility in a way that reflects the improvement in application quality. In practice, these utility values are often just estimates since measuring the increase in application quality is difficult.

To put a code block into an energy level, the developer has to place it into a conditional statement that checks if the level is currently active. The lowest energy level l_0 is always active and is declared implicitly; it includes all code that has not been added to any other level.

The abstraction of energy levels nicely fits modular development in component-oriented languages like nesC [Gay et al. 2003]. If an application consists of several nesC components which define their own energy levels, it might be undesirable that each of these levels can be deactivated separately. For example, code in one component might depend on functionality of another one's (higher) levels. Therefore, the developer can group them with a "wiring"-like mechanism. This is analogous to the

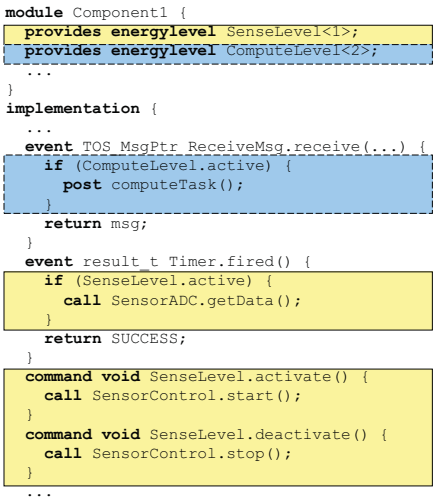


Fig. 2. Code example for energy levels

wiring of interfaces in nesC. Such combined energy levels are always active at the same time. Similar to the energy levels of individual components, the application itself forms its own energy levels. These energy levels are created by combining the energy levels of the application's components. In Fig. 1 the arrows show this mechanism. For instance, the energy levels of Component 3 and 4 are always active at the same time since they are combined in Component 2. Furthermore, the figure shows the overall energy levels of the application $(l_0, \ldots, l_3)$. This application can deactivate functionality for data analysis, storage, and sensing if necessary. Forwarding functionality, however, is placed on the lowest level l_0, which is always active.

Besides just combining energy levels, it is also possible to interleave the levels of different components. For example, in Fig. 1, Level 1 of Component 2 is mapped between the levels of Component 1 in the application. The only constraint is that it is not possible to change the order of the levels of a single component; doing so could break assumptions in the code.

By connecting all required levels to the lowest level l_0, the developer has full control of which energy levels have to be always active for the current application. Therefore, a component developer can create reusable components with many energy levels that are only deactivated if an application wires them to optional ones.

Compared to other programming languages for energy-aware applications such as Eon [Sorber et al. 2007], the model of energy levels has the advantage of its simplicity. With its direct integration in a widespread programming language, we feel that it is easier to learn than a separate coordination language like the one introduced by Eon. Furthermore, because of this integration, *Levels* does not have to resort to a data flow model but supports all interactions between components available in nesC.

2.2.1 *Syntax.* Our implementation integrates *Levels* into the nesC programming language. However, it would also be possible to add its abstractions to other lan-

guages. Fig. 2 shows example code that implements two energy levels. Their respective code is highlighted with the boxes. The numbers at the end of the declarations in the module header refer to their local order, which is not necessarily the global level number in the application.

To add code to an energy level it simply has to be called from within an "if" statement that checks if the level is active. Here the energy of asynchronously executed code like "computeTask" is attributed to the level from which it is called. Finally, the "activate" and "deactivate" functions allow the component to adjust to a new energy level by, for example, enabling a hardware device that is not needed in lower energy levels.

2.3 Energy Profiling

In order to correctly estimate the lifetime of the application, *Levels* has to know how much energy is consumed by each optional code block defined in energy levels. Getting this information on the sensor node itself is not possible since the energy consumed in individual blocks of code is too small to be accurately estimated using the node's built-in voltage sensor. Therefore, we make use of the fine-grained energy models available in simulators.

It should be noted that because of hardware differences the energy consumption of different nodes varies slightly [Landsiedel et al. 2005]. Currently, profiling can only achieve optimal results if the energy model of the simulator is calibrated to each node. Therefore, from our perspective, an important design requirement for future sensor nodes is that they should be created from parts which show only little variations in energy consumption.

Other approaches for energy-aware applications [Flinn and Satyanarayanan 1999; Zeng et al. 2002; Sorber et al. 2007] do not leverage simulation data. We opted for this approach since simulation is used frequently as a part of the sensor network development process anyway and since getting these numbers at runtime is hard or even impossible without additional hardware. In addition, compared to real measurements with instrumented sensor nodes and lab equipment, simulation has the advantage that the additional effort for the application developer is small.

Our energy profiling approach allows reusing code from *nCUnit*, a unit testing tool for sensor networks similar to JUnit. The only change needed for reusing unit testing code is to tag all relevant functions in the test driver with an "@energy" attribute that tells our build system which functions should be used to measure energy consumption. A pre-compiler generates calls for all measurement functions tagged with the "@energy" attribute. The parameters of this attribute specify the name of the function that should be profiled.

For each measurement function the energy profiler is executed several times, where – in order to avoid side-effects – each simulation only calls a single measurement function once. The profiler starts two separate simulation runs for all energy levels and each of their optional code blocks: one with a short duration t_1 and one with a longer duration t_2.

These energy measurements allow the system to compute two kinds of energy consumption for each of these code blocks: energy that is consumed once (i.e., when the code is executed) and energy that is consumed continuously (i.e., by changing the state of a hardware device). For example, sending a message requires energy

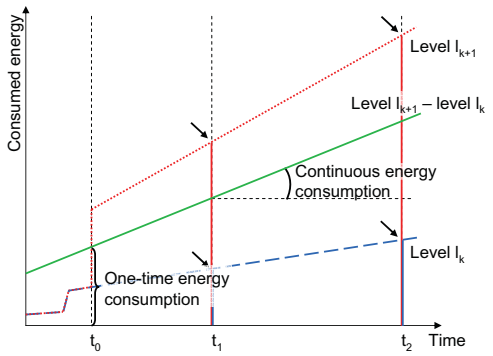


Fig. 3. Computing the energy consumption of a code block

only once whereas turning on sensors leads to a change in continuous energy consumption. In addition, the measurements allow to remove the overhead introduced for setting up the test case.

Fig. 3 shows how this computation is done for the case when the function under measurement defines at most one optional code block for each of the energy levels involved. The function is called at t_0. To get the increase in energy consumption of level l_{k+1}, four measurements are necessary: the energy consumptions of the function under measurement in levels l_k and l_{k+1} at both t_1 and t_2 (see the arrows in the figure). Then the difference between the two levels is computed by subtracting their values. From the resulting points, the slope of the energy difference, which corresponds to the change in continuous energy consumption, and the one-time energy overhead at t_0 can be computed.

This computation assumes that continuous energy consumption is constant. We expect this to be true on average for sufficiently long simulation durations. For example, if a timer is activated to periodically execute some code or if the sensor board is turned on, average energy consumption will be constant and the overall energy consumed will increase linearly with time.

We do not measure the energy consumed by the default energy level l_0 and rather compute this value at runtime by subtracting the energy of all other levels from the overall energy consumed. This decision helps to keep the runtime overhead of *Levels* small since this level would be present in every single function. Furthermore, because there are no optional code blocks in level l_0, profiling could be done only at a coarse granularity and, therefore, would probably be inaccurate.

Our profiling approach has several advantages. First, reusing unit testing code ensures that the code is executed in a controlled setting where, for example, messages from other nodes, unexpected sensor readings, or interactions with other components do not alter the application flow. Second, these measurements do not only include the energy spent by the CPU to run the code under test but also the energy consumption of other hardware like the radio or flash memory chips. Finally, unlike existing energy profilers [Landsiedel et al. 2005], which map energy consumption to code blocks, or systems monitoring energy consumption at runtime [Dunkels et al. 2007; Jiang et al. 2007; Dutta et al. 2008] our approach allows to include the energy consumption of asynchronously executed code (e.g., TinyOS tasks,

timers, split-phase events) in the measurement. This is important to get the total energy consumption originating from the code block: Since this code is executed from a certain energy level, its energy consumption has to be attributed to that level.

2.4 Runtime System

In this section, we describe the runtime system of *Levels*. The runtime system has three tasks: estimating the remaining energy from voltage readings, attributing energy consumption to energy levels, and adjusting the active levels at runtime. Besides these tasks, the runtime system also executes the distributed assignment algorithm described in Section 3.

2.4.1 *Battery Model.* To estimate the remaining lifetime it is necessary to know how much energy is left in the battery. For this purpose we have built a simple battery model that maps voltage values to the remaining usable battery capacity.

Creating such a model has not been the main focus of our research. Using the voltage sensor present on many sensor nodes, our battery model simply maps voltage readings to the remaining energy. Therefore, unlike Eon [Sorber et al. 2007], this model allows us to implement our system with unmodified, off-the-shelf sensor nodes. We created our battery model for the specific type of alkaline-manganese dioxide batteries [Duracell Batteries 2001] that we use in our experiments.

We opted for a simple but efficient model: for each distinct voltage reading we store the average remaining energy of this value in program memory. Because typical sensor network platforms like Mica2 nodes are not equipped with a voltage boost converter [Polastre et al. 2005], the current draw I depends linearly on the battery voltage U. Thus, the resistance R remains constant. From $E = U \cdot I \cdot t$ and $R = \frac{U}{I}$ the effect on energy (and power) is quadratic: $E = \frac{U^2}{R} \cdot t$. However, when creating our battery model, we assumed a constant voltage $U_{const} = 3\,V$ for the computations. Therefore, instead of mapping the actual energy E to the voltage readings, our battery model and all energy values in the rest of this paper refer to values for $E \cdot \frac{U_{const}^2}{U^2} = \frac{U_{const}^2}{R} \cdot t$. This simplifies computations at runtime greatly because the energy consumption of a code block can be assumed to be independent of the current supply voltage of the sensor node. Nevertheless, using the same approach in the creation of the model and at runtime leads to consistent results that allow for accurate computations.

Fig. 4 shows the discharge behavior of three batteries. Although there are some differences, the curves are virtually equal when the batteries are almost empty. Especially there a good energy estimation is important to accurately meet a lifetime goal.

At runtime, we map each voltage value to the average remaining energy of several such curves. Since the relationship between voltage and the remaining energy is not linear, the differences in energy values between two consecutive voltage readings can vary significantly. This affects the accuracy of the mapping. Similar differences can exist between the models of several batteries, especially close to their full capacity. Therefore, we make this expected error available at runtime. This makes it possible to defer computations until significantly more energy than this error estimate has been consumed; hence the influence of the inaccuracies is reduced.

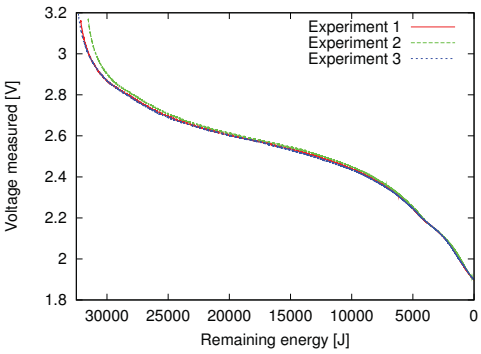


Fig. 4. Battery discharge characteristics from three experiments

We use this battery model not only for getting the remaining energy but – by subtracting the current value from the one of the last computation – also for getting the total energy consumed in that time period. However, probably better accuracy could be achieved here by using an approach based on special measurement hardware [Jiang et al. 2007; Dutta et al. 2008] or software counting the time spent in different hardware states [Dunkels et al. 2007]. However, hardware-based approaches can increase the cost of the sensor node by up to 100 % [Jiang et al. 2007] and are, therefore, not suited well for low-cost sensor networks. We did not apply a more complex software-based approach because we felt that this would unnecessarily increase runtime overhead. Nevertheless, these are viable alternatives but they cannot substitute energy profiling which would still be necessary to correctly attribute energy consumption to different levels. If an approach that uses a switching regulator were applied [Dutta et al. 2008], the computations of our system would have to be adjusted to the constant voltage it provides.

2.4.2 *Attributing Energy Consumption to Energy Levels.* The runtime system is responsible for attributing energy consumption to energy levels. First, it is called whenever an optional code block is about to be executed. It then checks if the energy level is active and adds the energy consumption of this code block to the energy consumed by that level. Second, every few seconds, it adds up the energy that has been consumed continuously in the current interval. Finally, every few hours, it computes the optimal level assignment for the time remaining.

If an optional code block of an energy level is about to be executed, the system checks if the level is active. The runtime system uses the data obtained from energy profiling (see Section 2.3) and adds the energy consumption of the current block to the overall energy consumed by the level. If a code block of level l_i is reached by executing code belonging to level l_j with $j > i$, the system correctly attributes the energy consumed by this code to level l_j. In addition, it updates continuous energy consumption. The same information is also updated for blocks belonging to the next higher energy level, which is actually not executed. This way, the system can predict the energy consumption after increasing the current level.

As already mentioned, for each energy level, *Levels* keeps information about its continuously consumed energy, e.g., for a hardware component that has been en-

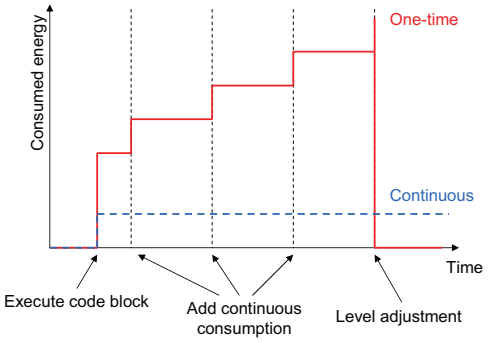


Fig. 5. Accumulating the energy consumed by a level

abled in an energy level. The runtime system periodically adds this energy that has been consumed in the last few seconds to the one-time energy consumption of the code. This approach provides finer granularity and, therefore, better accuracy than doing this only when computing the energy level assignment. In addition, it minimizes overhead because it requires less state and computational resources compared to calculating this data whenever continuous energy consumption changes.

Fig. 5 summarizes how the runtime system computes the energy consumed by an optional code block. When the code block is executed, both one-time and continuous energy consumption are updated. The system then periodically adds continuous energy consumption to the energy consumed by the level. After some time, this energy consumption is reset when computing a new level assignment.

2.4.3 *Adjustment of Active Energy Levels. Levels* uses the information about the energy consumption of energy levels to periodically adjust the currently active level. In each adjustment, it tries to maximize the utility of the energy levels for the time remaining while meeting the lifetime goals. Formally, this corresponds to the following optimization problem. Given the current lifetime t, the total required lifetime T_{req}, the remaining energy E_{rem}, and the energy levels $l_0, \ldots, l_{n-1}$, which have the utility values $u_0, \ldots, u_{n-1}$ and consume $P_0, \ldots, P_{n-1}$ energy units per time interval, find the durations $t_0, \ldots, t_{n-1}$ that maximize the utility of the energy levels for the remaining lifetime $T_{rem} := T_{req} - t$:

$$\begin{aligned} &\text{maximize} && \sum_{i=0}^{n-1} u_i \cdot t_i \\ &\text{subject to} && \sum_{i=0}^{n-1} t_i = T_{rem} \\ & && \sum_{i=0}^{n-1} P_i \cdot t_i \leq E_{rem} \\ & && t_0, \ldots, t_{n-1} \geq 0 \end{aligned}$$

The first equation formalizes the maximization of the utility over time. Each t_i corresponds to the duration for which Level l_i is the highest active level. The constraints then specify that the still needed lifetime has to be met and that enough energy has to be available. Using a linear equation for the energy constraint is only possible because our battery model returns the energy for a (hypothetic) constant voltage instead of actual energy values. Finally, the last equation excludes solutions with negative time durations.

The optimization problem can be solved using well-known algorithms from linear programming [Chvátal 1983]. In our implementation, we use the Simplex algorithm, which is the standard method for solving such problems. Since our implementation uses efficient fixed point arithmetic, the computational overhead of this algorithm is small. Furthermore, we limited the overhead by defining a maximum number of iterations after which the algorithm aborts even if it has not found the optimal solution yet. This limit is reached only seldom in practice, however. As we show in the evaluation, the computational overhead can almost be neglected even on resource-constrained sensor nodes (see Section 5.3).

The results of the optimization are the t_i values that specify for how long each energy level should be the highest active one. Because there are just two constraints (in addition to the exclusion of negative values), the result of the optimization has at most two non-zero time intervals. Therefore, independent from the total number of energy levels at most two of them will be activated. We leverage this property in our distributed assignment algorithm (see Section 3.2.2).

The system tries to compute a new level assignment periodically with a low frequency (e.g., every two hours). Repeating this computation is necessary since energy load may vary over time and because of possible inaccuracies in previous adjustments. However, due to the discharge characteristics of batteries the inaccuracies of the measurement might exceed the actual energy consumed, especially for low-power applications. In this case it is not possible to compute meaningful results. Therefore, we use the expected accuracy from the battery model at runtime: only if the energy consumed by code in energy levels is sufficiently big, the algorithm tries to compute a solution. Otherwise, it waits until the next measurement. Although this reduces the agility of the system, it helps to obtain correct results. In addition, to further reduce the fluctuations because of inaccurate measurements, we use a moving average to smooth the energy values used for computation.

Moreover, to deal with inaccurately estimated remaining battery capacities and with possibly varying load within an energy level, *Levels* adds a safety factor to the lifetime still required in order to make sure that the node can meet its lifetime goal. This design decision leads to the side effect that the average level achieved is slightly below the optimum because the lifetime goal is usually exceeded. We opted for this conservative policy to ensure that no node runs out of energy early. Furthermore, as the safety factor depends on the remaining lifetime required, this issue is addressed by periodically recomputing the level assignment. The node will – in later computation rounds – switch back to higher levels if sufficient energy is still available.

To minimize the complexity of the runtime system, *Levels* does not change energy levels between these computations; it makes use of just one of the levels from

the result and switches to the highest level returned. This allows us to consider always the most recent information about energy consumption before switching to another level. Depending on the energy consumption of the application and the current accuracy of the battery model, several tries might be needed until the level assignment can be recomputed. Therefore, a level is selected only if the algorithm expects to be executed again before the computed time duration. Otherwise, *Levels* switches to the next lower level in the solution.

3. DISTRIBUTED ASSIGNMENT OF ENERGY LEVELS

In applications such as volcano monitoring [Werner-Allen et al. 2006] and structural health monitoring [Kim et al. 2007] the single largest energy consumer is the sensor board. Even the energy consumed for radio communication is much smaller. If the sampling code is put into an energy level and all nodes sample at the same time, their energy load will be the same. Therefore, if each node assigns its energy levels independently, all nodes will switch energy levels almost synchronously. Over time the overall quality of the network will be very high in the beginning before then becoming very low. Many applications, however, benefit if the overall average quality of all nodes remains constant and if the nodes offering different energy levels are distributed throughout the network. In the example applications, this means that there are always some nodes that gather data and that these nodes can be found throughout the network.

Levels provides two mechanisms to address this problem. The first one makes use of randomization whereas the second one is based on a distributed coordination algorithm. Both of them still rely on the local maximization of the energy levels and simply determine when to activate the levels from this local result. Therefore, the optimization described in Section 2 is still done on each node, and all nodes still maximize their own utility value; the only difference of the approach presented in this section is that the energy levels returned by the optimization are activated at a different point of time on different nodes.

Instead of always activating the highest energy level that is part of the local optimization result, our first approach *randomly* selects an energy level from this result. This energy level is activated first; the other ones that are part of the local optimization will be selected in later rounds. Although there might be some slight variations in overall application quality, this mechanism is well-suited for low-power applications since it does not require any coordination among nodes. It achieves an approximately uniform distribution of energy levels throughout the network – given that all nodes have similar local optimization results.

The second approach performs some coordination among neighboring nodes. Using also the local optimization results, it creates a level assignment such that the deviation from the average energy level of all nodes is minimized. Minimizing that deviation helps to reduce the fluctuations in overall application quality. Therefore, depending on the definition of energy levels, this approach increases the probability that there are always some nodes present that, for example, sample data.

This coordination is done in a completely distributed way. As we show in the evaluation, both the computational and the communication overhead are minimal. The result is not necessarily a network-wide uniform distribution; it rather takes

into account each node's neighborhood. Only the neighboring nodes have to be considered since they observe the same area.

Apart from determining the nodes' energy levels, our coordination approach can be used to compute an activation schedule for the nodes. This is useful in applications where some nodes are redundant and can be deactivated for some time. In our view, such a deactivated node does not participate in the application at all, i.e., it stays in the CPU's lowest power state and wakes up only for coordinating schedules. Only after becoming active, it contributes to the application in its usual way, e.g., by sensing and forwarding data. An active node may still have different energy levels and may still make use of energy-efficient techniques such low power listening [Polastre et al. 2004].

In many applications, redundant nodes can be added to increase network lifetime and the resilience to node failures. Since these nodes do not have to be active for the complete network lifetime, they have more energy available for each active interval and can, therefore, switch to higher energy levels. Furthermore, network lifetimes that are longer than the maximum lifetime of each node become possible. The mechanisms introduced for balancing energy level assignments can also be used to compute the activation schedules of redundant nodes. This part of *Levels* is optional; if no redundant nodes are available in the network, only energy level assignments will be coordinated.

Determining from all nodes the set of active ones is closely related to existing coverage and topology control algorithms. However, we only deal with the problems addressed by these algorithms implicitly through distributing active nodes in the network and controlling network density. Therefore, we cannot give any guarantees about coverage and network connectivity. Unlike our approach, the vast majority of coverage and topology control algorithms tries to maximize network lifetime [Cardei and Wu 2006; Huang et al. 2006; Cerpa and Estrin 2002; Ye et al. 2003]. For *Levels* a different optimization goal is needed because we assume that the required lifetime is given by the developer.

The distributed coordination approach is described in the rest of this section in more detail for both the problem of determining the schedules when to activate nodes and how to balance energy levels.

3.1 Problem Description

This section describes the problem of finding an optimal schedule to activate nodes and assigning their energy levels. Basically, this problem corresponds to finding a permutation of the assignment that minimizes the deviations of the number of active nodes or the energy level, respectively, from their average over time. If this deviation is zero, network-wide application quality will be constant. As we describe below, both the number of active nodes and the energy level have an impact on application quality.

We assume that the user specifies the individual required lifetime T_{req} besides the lifetime required for the whole network. This value can be shorter than the network lifetime if some nodes are redundant. T_{req} can be given either directly, or it can be computed from the number of nodes in the neighborhood, the user's desired number of active nodes, and the network lifetime.

In addition, our algorithm assumes that nodes are not mobile. This is necessary

since it computes the schedule for a longer time based on information about the nodes present in the vicinity. This would not be possible if the nodes moved. Moreover, some (loose) synchronization among nodes is needed because nodes have to transmit their results to their neighbors. If a node is deactivated, it has to wake up to receive these messages from its neighbors. A node that receives such a message before doing its own computation already considers the new values.

Furthermore, we continue to assume that the developer specifies the overall required lifetime for the complete network. From that number and the currently achieved network lifetime the remaining required network lifetime $\overline{T}_{rem}$ can be computed. If the remaining lifetime of node j is used, this is expressed as $T_{rem}(j)$. This function is defined for each node with $1 \leq j \leq R$, where R is the number of nodes to consider (e.g., the nodes in the radio neighborhood, including the current node).

Based on the lifetime of the nodes, a schedule can be computed to activate and deactivate them over time. For this purpose, we use an approach that tries to keep the number of active nodes constant, i.e., it tries to minimize their deviation from the average over time. Furthermore, as a secondary goal, not just the number of active nodes should stay close to its average, but also the utility of the currently selected energy levels. Determining the set of active nodes is the primary goal since we assume that variations in energy level assignments are less critical for overall application quality than variations in the number of active nodes. The input for balancing energy levels is the result of each node's local maximization of the utility since each node should still provide the best quality for its required lifetime. Only if both the number of active nodes and the average utility value are (roughly) constant, the application quality will not vary over time.

Instead of computing the activation schedules separately from the energy levels, an alternative approach would be to interpret deactivating a node as an additional (lower) energy level. We have not selected that approach because a deactivated node is different from a node that just reduces its functionality. Therefore, it would be difficult to increase the priority of the problem of computing the activation schedule.

3.1.1 *Computing a Schedule for Activating Nodes.* To compute an optimal schedule for activating nodes, we express the schedule of each node as a string of 0 and 1. Each character corresponds to a time interval. If it is 0, the node is turned off in this interval whereas it is turned on if the corresponding character is 1. The length of the string corresponds to the total remaining network lifetime $\overline{T}_{rem}$ and the number of 1 characters is the remaining lifetime of the node (T_{rem}). Such a string is given for each node. A valid schedule for the network is a combination of the schedules of individual nodes.

The schedule of node j at time interval i is defined by the following function:

$$Active(j,i) := \begin{cases} 1 & \text{if node } j \text{ is scheduled to be active in interval } i \\ 0 & \text{otherwise} \end{cases}$$

To solve the problem of keeping the number of nodes constant, each node has to know R_{avg}, the average number of nodes in its neighborhood that are active in

1:	Node 1: 011 Node 2: 001	**4:**	Node 1: 101 Node 2: 001	**7:**	Node 1: 110 Node 2: 001
2:	Node 1: 011 Node 2: 010	**5:**	Node 1: 101 Node 2: 010	**8:**	Node 1: 110 Node 2: 010
3:	Node 1: 011 Node 2: 100	**6:**	Node 1: 101 Node 2: 100	**9:**	Node 1: 110 Node 2: 100

Fig. 6. Combinations of activation schedules for two nodes

each time interval. This information can be computed by dividing the sum of the required node lifetimes of neighboring nodes by the remaining lifetime required for the network:

$$R_{avg} := \frac{\sum_{j=1}^{R} T_{rem}(j)}{\overline{T}_{rem}}$$

A solution that provides constant application quality always has a fixed number of nodes active. To minimize variations, this number should be as close as possible to R_{avg}. The deviation from R_{avg} can be computed in the following way:

$$\Delta_{active} := \sum_{i=1}^{\overline{T}_{rem}} \left| \sum_{j=1}^{R} Active(j,i) - R_{avg} \right|$$

For example, Fig. 6 shows all combinations of schedules for two nodes. The network lifetime is assumed to be three time intervals. Node 1 can be active for two intervals whereas Node 2 can be active for just one time interval. In this example R_{avg} is 1, i.e., a solution is optimal if exactly 1 node is active in every time interval. Therefore, there are three optimal solutions (solutions 3, 5, and 7).

3.1.2 *Balancing Energy Level Assignments.* When energy level assignments are coordinated, each node continues to solve the optimization problem of Section 2.4.3. Therefore, it still optimizes its average utility over its lifetime. The only difference is that the point of time when to select each level assignment is determined by taking into account the assignments of other nodes.

Just like a schedule of active nodes, an energy level assignment can be expressed as a permutation of a string of level numbers. If the node is scheduled to be active in the respective time interval, the value corresponds to the number of an energy level. Otherwise, the value is undefined.

Again the goal is to minimize the deviation from each node's local average. Now, however, the average utility of the energy levels has to be considered. Therefore, we define the function $Utility(l)$ that maps energy level numbers to their corresponding utility values.

The energy level of node j at time interval i is defined by the following function:

$$Level(j,i) := \begin{cases} \text{level assigned at time } i & \text{if } Active(j,i) = 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

3a:	Node 1: -12 Node 2: 1--	**5a:**	Node 1: 1-2 Node 2: -1-	**7a:**	Node 1: 12- Node 2: --1
3b:	Node 1: -21 Node 2: 1--	**5b:**	Node 1: 2-1 Node 2: -1-	**7b:**	Node 1: 21- Node 2: --1

Fig. 7. Combinations of energy level schedules for two nodes

Using these two functions and the function $T_{rem}(j)$, which returns the remaining required lifetime of node j, the average utility of the nodes in all time intervals can be computed:

$$U_{avg} := \frac{\sum_{j=1}^{R} \left(\sum_{i=1}^{\overline{T}_{rem}} Active(j,i) \cdot Utility(Level(j,i)) \right)}{\sum_{j=1}^{R} T_{rem}(j)}$$

The numerator of this fraction sums up all utility values from the energy levels assigned to the nodes. The denominator, in contrast, computes how long all nodes are active throughout the network lifetime. If that number is 0, no level assignment is needed since no node will be active in the time remaining.

Again, the deviation from the average is to be minimized in order to provide constant application quality. This deviation can be computed by subtracting the overall average utility U_{avg} from the average utility of the active nodes in each time interval:

$$\Delta_{levels} := \sum_{i=1}^{\overline{T}_{rem}} |\delta(i)|$$

$$\text{with } \delta(i) := \begin{cases} \frac{\sum_{j=1}^{R} Active(j,i) \cdot Utility(Level(j,i))}{\sum_{j=1}^{R} Active(j,i)} - U_{avg} & \text{if } \sum_{j=1}^{R} Active(j,i) \neq 0 \\ U_{avg} & \text{otherwise} \end{cases}$$

We try to find a permutation of level assignments with the minimum value for Δ_{levels}. Only if a node is active, it can be assigned an energy level. In addition, to give preference to the first problem, the schedule of when to activate a node has to be optimal. However, such an optimal schedule is not necessarily unique. Therefore, for the optimum solution, it might be necessary to change the activation schedule if this could improve Δ_{levels}. A different (optimal) activation schedule might lead to different energy level assignments since other nodes can be active at the same time.

Fig. 7 continues the example from Fig. 6. It assumes that Node 1 can switch for one time interval to Level l_1 and for another one to Level l_2. Node 2, in contrast can only select Level l_1. If the utility values are equal to the level numbers, the average utility U_{avg} in this example is $\frac{4}{3}$. This value cannot be achieved exactly because energy levels for fixed time intervals are to be assigned. The best solutions have a deviation from the average level of $\frac{1}{3} + \frac{1}{3} + \frac{2}{3} = \frac{4}{3}$, where the $\frac{1}{3}$ terms refer to the time intervals in Level l_1 and the $\frac{2}{3}$ term to the time interval in Level l_2.

```
While T̄_rem > 0
  Listen for schedules from neighbors and wait
    On receive:
      Store schedule received from neighbor
  Compute local schedule
  Broadcast local schedule to neighbors
  Wait until next computation round
```

Fig. 8. Algorithm for the distributed optimization

As the figure shows, all three optimal assignments from Fig. 6 lead to the same deviation. In fact, even creating a different permutation over the level assignments (the variations a and b of each solution) does not change the deviation in this simple example.

3.2 Realization on Sensor Nodes

This section describes how the problems introduced in Section 3.1 can be solved on the sensor nodes. First, it shows how the solution can be approximated in a completely distributed way that requires the nodes only to send minimal amounts of data. Second, it presents a greedy approach that, in combination with the distributed algorithm, efficiently finds a solution.

3.2.1 *Distribution in the Network.* Basically, to implement an assignment algorithm for the sensor network there are two alternatives that do not require a global view of the network: having a cluster head assign schedules to its neighbors and computing a local schedule on each node while using the neighbors' schedules as constraints.

Although the first solution is promising, it has the disadvantage that some nodes – i.e., the cluster heads – have to do significantly more computation and communication than the others. Therefore, their energy budget is burdened most. This problem could be alleviated only by switching the role of a cluster head at runtime, which would probably require to change the boundaries of clusters as well. We think that this would add significant overhead to the system. Furthermore, computing an exact solution to the problem for all nodes of the cluster would probably increase complexity beyond the capabilities of the sensor nodes. Because of that, they could only approximate an optimal solution.

Therefore, we selected the second alternative, where each node computes its own schedule. Although this approach might not find a globally optimal solution (i.e., the solution that minimizes the deviation for all nodes in the network), it schedules the current node optimally with respect to the schedules of neighboring nodes. In addition, it has the advantage that its overhead is comparatively small: each node only has to compute its own schedule and broadcast it to its neighbors. Depending on the number of energy levels, it is sufficient to store and transmit a few bits for each time interval. Moreover, to avoid too frequent level changes and to further reduce the overhead, longer intervals than those for the local optimization in Section 2.4.3 can be used – at the cost of reduced agility. If these longer intervals are used, a node can send its schedule in a single message because a complete schedule requires just a few bytes.

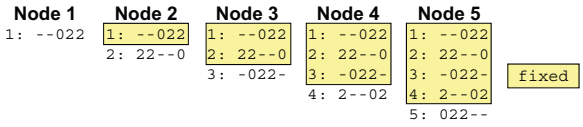


Fig. 9. Distributed computation of energy levels

Fig. 8 shows an overview of our distributed algorithm. It is repeated for each interval of the distributed optimization. First, the nodes collect and store the schedules of their neighbors before doing their own computation. These schedules from neighbors are fixed constraints for each node's optimization. Neighboring nodes from which no schedule has been received so far are not included in the optimization. Therefore, the only schedule that can be modified is the one of the local node itself. This reduces the complexity of the computation greatly. In addition, this approach considers the specific neighborhood of each node. However, since only the local schedule can be modified, the solution is not globally optimal. After the computation, the node sends its schedule to its neighbors and waits until the next round.

Even though this optimization is done for just a single node, it is too expensive for resource-constrained sensor nodes to compute all possible permutations. Therefore, we pursue a different approach that greedily activates nodes.

Each node repeats its assignment periodically in order to deal with changes in the local optimization results and with failures of other nodes. In later rounds, previous results from all neighbors can be used to further improve the assignment even if a node has not updated its schedule for the current round yet.

Just like the randomization approach, this optimization is not fixed to specific levels: if during a later local optimization an energy level is no longer part of the result and if the distributed assignment has not been updated yet, a node might select another energy level from its local optimization instead. Therefore, there can be some variations in energy level assignments that are not detected by the distributed computation.

Fig. 9 shows an example of how the distributed computation is done. The example assumes that all five nodes can communicate with each other and that they do the optimization in the order of their IDs. Furthermore, the network lifetime is assumed to be five time intervals and each node can be active for three of them (two in Level l_2 and one in Level l_0).

Since Node 1 just has to consider its own schedule for the first computation round, any assignment is as good as any other one. After the computation, it sends its schedule to the other nodes in its neighborhood. Node 2 uses this schedule when computing its own activation times and energy level assignments but it does not modify the schedule of Node 1. Then it locally broadcasts its own schedule to the other nodes. Likewise, the remaining nodes compute their schedules using those of the first ones as input. In later rounds, all nodes will consider the schedules previously received from all neighbors.

In this example the results for both the activation schedule and the level assignment are optimal. In each time interval three nodes are active, which is also R_{avg}, the number that the algorithm has tried to achieve. Likewise, the average level is

```
// initialization
Compute R_avg
For τ = 1 to T_rem
  Set activation schedule in interval τ to 0
Min_Deltas = {}

// determine intervals with minimum deviations
For τ = 1 to T_rem
  Set activation schedule in interval τ to 1
  Compute Δ_active
  If |Min_Deltas| < T_rem
    Add (τ, Δ_active) to Min_Deltas
  Else if Δ_active smaller than largest value in Min_Deltas
    Remove element with largest value from Min_Deltas
    Add (τ, Δ_active) to Min_Deltas
  Else if Δ_active equals largest value in Min_Deltas
    Randomly decide if the entry in Min_Deltas should be replaced
  Set activation schedule in interval τ to 0

// activate node in these intervals
For all (τ, Δ_active) ∈ Min_Deltas
  Set activation schedule in interval τ to 1
```

Fig. 10. Algorithm to compute the local schedule

in this example equal to the optimum U_{avg} of $\frac{4}{3}$.

3.2.2 *Schedule Assignment on Each Node.* To compute an optimal schedule of a single node, it is not necessary to check all permutations of its schedule. In fact, it is sufficient to use a greedy approach that selects those time intervals (one after the other) where the deviation from the average is the smallest one. Fig. 10 shows this algorithm that computes the activation schedule of a node.

The algorithm first calculates the average R_{avg} as described in Section 3.1. This computation already assumes that the node is turned on for its required lifetime even though it has not been assigned time intervals yet. Thus, R_{avg} corresponds to the actual target value.

The algorithm initializes its variables to deactivate itself in all time intervals (all values set to 0). Then it computes the deviation values from R_{avg} if the node is active in just one interval. This computation is done for all time intervals, and the node stores the values with the minimum deviations in Min_Deltas. These numbers refer to the intervals when the node should be active. If there are several possibilities which intervals to select, i.e., if the current Δ_{active} is equal to the largest element in Min_Deltas, the algorithm randomly decides which interval should be chosen. This results in a better distribution over time than always selecting the first one, for example. Finally, the node schedules itself to be active in the intervals corresponding to the values in Min_Deltas.

For example, if in Fig. 9 Node 5 tries to assign its active time slots, it first computes the overall average of active nodes per time interval, which – in this example – is 3. It then assumes that it is active in a single time slot and computes the deviations from the average. Regarding the other nodes, two of them are active

in the first three time intervals whereas already three nodes are active in the last two intervals. If Node 5 is activated in one of the first three intervals, the overall deviation Δ_{active} is 2. If it is activated in one of the last two intervals, however, the deviation from R_{avg} is 4. Therefore, since the node is required to be active for three time intervals, it schedules itself to be active in the first three intervals.

This algorithm always finds an optimal solution subject to the schedules of the neighbors and considering the fixed-length time intervals: If the node selects a time interval with the smallest deviation from R_{avg}, this interval has to be part of an optimal solution since it corresponds to the summand of Δ_{active} with the minimum values. Because all the summands of Δ_{active} are non-negative, the sum itself becomes minimal if all of its summands are minimal. The only way to influence the deviation from the average is to make the node active. In addition, the node has to be scheduled for exactly its remaining lifetime. Therefore, there is no other assignment that could lead to a smaller overall deviation.

The efficiency of this algorithm is much better than that of an algorithm that computes all permutations. In fact, it just has to compute $\overline{T}_{rem}$ values for the deviation. If the overhead of a straight-forward solution to maintain the list of time intervals with minimum deviation is considered, the overall complexity of this algorithm is $\mathcal{O}({\overline{T}_{rem}}^2)$. Compared to the number of permutations of the schedule, this is a significant improvement since that number depends on the factorial of $\overline{T}_{rem}$.

The problem of balancing energy levels can be solved analogously because the local optimization of Section 2.4.3 returns at most two energy levels with non-zero time durations in each solution. This directly corresponds to the problem of activating nodes with the two states “on” and “off”. Here, however, only time intervals are considered in which the node is scheduled to be active. In the beginning, all of these intervals are assigned the lower energy level from the local optimization result and the target value for the average utility U_{avg} is calculated. Then, the node computes the resulting deviation Δ_{levels} from U_{avg} if it assigns its higher energy level to each interval, and selects the intervals with the minimum deviations.

Using this algorithm and the distributed computation outlined in the previous subsection, the computational overhead for each node is minimal. Therefore, as we show in the evaluation, even resource-constrained sensor nodes can optimize their own schedules with respect to those of their neighbors.

Although this approach finds an optimal solution for each of the two sub-problems subject to their constraints, its overall solution might not be the global optimum for all nodes and for both sub-problems combined. First, because of the distributed computation a node cannot modify its neighbors' schedules. Therefore, the solution might not be the network-wide optimum. Second, because assigning the active time intervals of nodes and balancing the energy levels are solved independently from each other, a node might not be active in time intervals where it could approximate U_{avg} more closely even if such an activation schedule might also be optimal. For example, in some time intervals, only those nodes with low energy levels left might be active. However, experiments have shown that this is not a problem in practice.

4. LIMITATIONS

In this section we describe some limitations of the approaches presented in this paper.

First, *Levels* assumes that discrete levels of functionality can be defined in the application. It is currently not possible to specify a parameter that, for example, sets the sampling rate to an arbitrary value. We did not include such a parameterization because we think that it would increase the computational effort at runtime significantly. However, it is possible to create several energy levels that adjust the sampling rate to pre-defined values when they are activated, for instance.

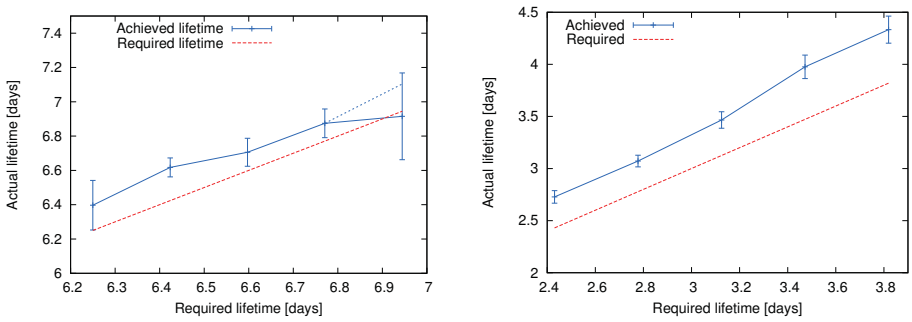
Second, if all nodes have to be constantly active and sample data in order to provide the functionality of the application, such functionality cannot be put in an energy level. One example for that kind of application is the localization of shooters [Simon et al. 2004] where the event to detect is infrequent and quick such that all nodes have to sample continuously and to be able to collaborate with neighboring nodes. Therefore, such an application can hardly benefit from *Levels*.

Third, our current implementation supports only applications that try to achieve a given lifetime with a fixed amount of energy. However, the abstraction of energy levels could also be applied to other optimization goals. For example, as alternatives, it would be possible to provide a given average utility while maximizing node lifetime or to ensure perpetual operation if the node makes use of energy harvesting.

Fourth, *Levels* assumes that the frequency of level adjustments is sufficiently greater than the frequency of events that influence energy consumption (e.g., volcanic eruptions [Werner-Allen et al. 2006]). Furthermore, it assumes that the average frequency of such events is approximately constant for the computation intervals. To ensure that these conditions are fulfilled, the developer might have to increase the length of computation intervals depending on the specific requirements of the application. Nevertheless, by periodically recomputing level assignments and by exchanging this information with neighboring nodes, *Levels* is able to deal with some variations in energy consumption over time.

Fifth, the energy levels defined by the developer are likely to be relatively coarse-grained. We opted for this solution because in our opinion it simplifies application development since the code can rely on other functionality in the same level. However, with a more fine-grained definition of optional functionality – which, in fact, would still be possible with our programming abstraction – much of the coordination effort described in Section 3 could be avoided: if there are more energy levels, it is more likely that a node can stay in a single level throughout its lifetime. Therefore, no coordination about when to activate which level would be necessary – given that there is actually such a single level of functionality and that energy measurements are accurate enough to safely activate it.

Finally, our distributed assignment approach deliberately does not address coverage or topology control problems [Cardei and Wu 2006; Huang et al. 2006; Cerpa and Estrin 2002; Ye et al. 2003]. This means it does not ensure that the sensor network constantly monitors a given area and that the network stays connected. It rather tries to provide a constant level of functionality in each neighborhood but cannot give any guarantees about that. Therefore, it cannot prevent temporal or spatial holes in sensor data, i.e., time periods without any sensor reading in some



(a) Experiments with Mica2 nodes (b) Volcano

Fig. 11. Average lifetime (including 95 % confidence intervals)

areas, for example. We made this design decision since the functionality of energy levels is application-specific: *Levels* does not know if an energy level includes sensing or networking code, for example. If an application needs stronger guarantees about the functionality of the network, it has to do its own application-specific coordination. For this purpose, the application can give hints about which energy level should be active. Our distributed assignment mechanism is just one option that we believe to be frequently required by applications.

5. EVALUATION

This section evaluates the benefits and overhead of *Levels*. For this purpose, we use both experiments with real sensor nodes and simulation.

5.1 Quality of Level Assignments

The quality of level assignments is shown best by comparing the lifetime achieved to the required lifetime. Ideally, the lifetime achieved should not be much longer than required because otherwise application quality could have been improved.

5.1.1 *Experiments with Mica2 Nodes.* First, we performed some experiments with real hardware using Mica2 sensor nodes. For this purpose, we created a simple application called Voltage that periodically sends messages with its current voltage reading and, on the highest energy level, toggles the node's LEDs every thirty seconds. Because of the time constraints of our experiments we intended to create a particularly energy-intensive application here. To prevent side-effects from slight variations in energy consumption, we calibrated the energy model used for profiling with a multimeter to the specific sensor nodes used and created a separate battery model for each node.

Fig. 11(a) shows the actual lifetime achieved by the motes when varying the required lifetime. We define as the lifetime the time a neighboring node was able to receive periodically transmitted packets, which were sent irrespective of the current energy level. In most experiments, the nodes met their lifetime goal. However, in the last experiments we ran – some of those with a lifetime of 10,000 minutes (6.94 days) – most nodes failed early although in previous experiments they accurately achieved this lifetime. For this time value these failures reduce the average

lifetime and increase the size of the confidence interval. We had purchased the batteries used in these (failed) experiments several months after the ones used to build the battery model. A detailed analysis of the recorded voltage readings showed that the nodes expected to have significantly more energy left than was actually available. We attribute this to differences in the properties of the batteries. In fact, after updating our battery model, a lifetime of 10,0000 minutes could be achieved again. This shows that estimating the remaining battery capacity from the voltage is extremely error-prone. Therefore, a good representation of the battery characteristics is needed for *Levels* to accurately meet lifetime goals. If the experiments that failed are not considered for this lifetime, the nodes would stay operational for 10,229 minutes (7.10 days) on average. This is shown with the dotted line segment in Fig. 11(a).

The lifetime achieved by the sensor nodes in all other experiments was between 1.1 % and 6.5 % longer than the lifetime requested. Considering variations in battery capacity and the fact that we used a fixed interval for level adjustments – which in this short-lived application resulted in a relatively small number of possibilities to adjust the level – these numbers are excellent. Although the variations due to external influences are slightly greater here, the results correspond to simulation results [Lachenmann et al. 2007]. Therefore, they show that the models used in the simulator capture the relevant factors of real deployments.

5.1.2 *Real-World Application.* In this section, we show how *Levels* can be applied to a real-world application. For this purpose, we selected monitoring of volcanoes [Werner-Allen et al. 2006]. In this application, there is usually no redundancy in the network topology, and the required duration of the experiment is known in advance. Replacing batteries is extremely difficult due to the inaccessible deployment location. Moreover, large parts of each node's energy are used to power the sensor interface board. Therefore, if a node stops sampling data but continues forwarding data from other nodes, its lifetime can be extended significantly and network connectivity can be preserved much longer.

As a concrete example of this class of applications, we chose the system used at Reventador ("Volcano") [Werner-Allen et al. 2006]. This system is a complex application that has been tested in real-world deployments. It stores sensor readings to flash memory, and the base station can then request stored data. In addition, Volcano includes an in-network detection of volcanic eruptions.

We use the Avrora simulator [Titzer et al. 2005] for this evaluation, which accurately emulates the behavior of Mica2 nodes. However, since this simulator did not include the custom sensor interface board used by this application, we had to add its energy consumption to the simulator's energy model. From the information available we assumed for the sensor board a current draw of 40 mA. Furthermore, Volcano has been originally created for Telos B nodes while the prototype implementation of *Levels* assumes the Mica family of sensor nodes. Therefore, we ported the application to this hardware family. However, in order to keep changes to the application small, we simulated a fictitious Mica2-like node that is like the Telos B nodes equipped with more RAM.

The battery voltage that the simulator makes available to the sensor nodes' voltage sensors has been recorded from individual batteries with the voltage sensor

of a real sensor node. In contrast, at runtime *Levels* uses a battery model based on the average of several such voltage traces (see Section 2.4.1). Therefore, this simulation setup corresponds to the situation of real sensor nodes.

In the deployment of the application [Werner-Allen et al. 2006], some batteries with higher capacities than those in our battery model were used. Therefore, our simulated lifetimes are significantly shorter than those reported there; this reduction in lifetime is not due to *Levels* and can also be observed when simulating the original application with the parameters of our batteries.

The behavior of Volcano depends on the eruptions detected by the sensor nodes. We simulated these eruptions at random intervals such that, on average, one event occurred every 30 minutes. However, like in the real deployment, not all of these eruptions were actually reported by the nodes if they, for instance, stopped sampling to transfer some data.

In its original version, Volcano does not include code to achieve a user-defined lifetime goal. Therefore, we specified some optional functionality using our energy level abstraction. Since sensing is the largest single energy consumer, we put this code into a separate energy level. If it is deactivated, the nodes turn off the energy-expensive sensor interface boards and stop analyzing, storing, and transmitting their data. However, they still fully participate in routing and, thus, forward data from other nodes.

We defined energy levels in two nesC modules that were then mapped to a single level in the application. Only minor changes to the existing code were necessary: about 20 lines of code had to be added or modified. Some larger effort was, however, needed to write the profiling functions since no suitable unit test drivers were available. The size of this module is less than 200 lines of code. In addition, we were able to copy almost the complete nesC wiring from the actual application and reuse it for energy profiling.

Fig. 11(b) shows the average lifetime achieved by this application. In total, we simulated 150 sensor nodes and none of them failed before its lifetime goal. However, since in this complex application the behavior of the nodes depends on network packets received and random events detected, future energy consumption cannot be accurately predicted from past data. This is shown with the confidence intervals in Fig. 11(b), whose sizes are between 3.6 % and 6.0 % of the lifetimes requested. In addition, *Levels* is not able to completely consume the energy kept as a safety buffer, and the nodes live on average 12.4 % longer than required. This number could be reduced, however, by adjusting some parameters of *Levels*. Nevertheless, considering the limited predictability of this application, even these results are encouraging.

5.2 Distributed Assignment

This section evaluates the behavior of our distributed coordination algorithm. The most important metric for this algorithm is the deviation from the average since minimizing this value is the overall optimization goal. Therefore, we run different algorithms and compare this value both for the number of active nodes and for the average energy level assigned.

First, we use the original approach from Section 2.4.3 that does not balance energy level assignments among nodes: It always starts with the highest level first

and switches to lower levels only later. In addition, for this evaluation we assume that it turns on all nodes for their required lifetime immediately after deploying the network.

Second, we use the randomization scheme from Section 3, which randomly decides which level from the local optimization result should be selected first. This approach does not require any additional optimization or coordination.

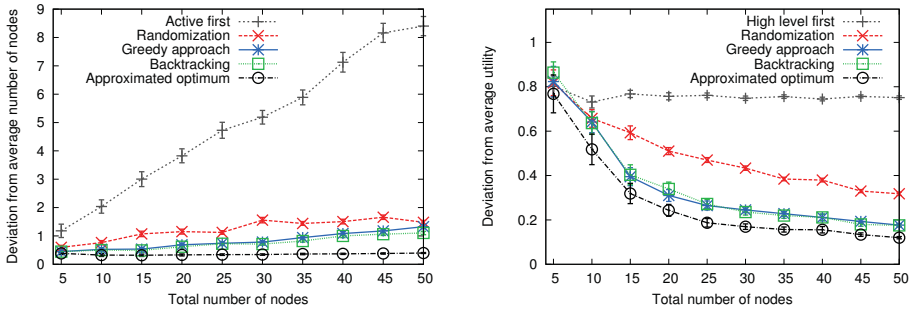
Third, the distributed greedy approach from Section 3.2.2 computes a solution that takes into account the schedules of the neighbors.

Fourth, as a first benchmark, we compute a solution using a backtracking algorithm in combination with our distributed assignment algorithm. Each node only can modify its own schedule; the schedules of its neighbors are fixed. Therefore, the solution found might not be the global optimum for all nodes. Since this computation is done for just a single node in each case, it is actually possible to compute all permutations of its schedule on a more powerful device like a PC. In addition, this approach adjusts the activation schedule if an equivalent solution exists that allows for a better level assignment.

Finally, as a second benchmark, we have approximated a solution that is optimal for the complete network using simulated annealing [Kirkpatrick et al. 1983]. Simulated annealing has been developed as a meta-heuristic for optimization problems. It has been inspired by annealing in metallurgy, where some material is temporarily heated again in its cooling process in order to increase the size of its crystals. Simulated annealing starts with a valid solution for the optimization problem and continuously modifies it. It continues with the new solution if it is better than the previous one. However, transferring the annealing concept, even an inferior solution can be selected with some probability. Therefore, simulated annealing avoids being stuck in local optima. Using a heuristic is necessary since the complexity for realistic problem sizes is too high to compute the exact optimum. However, for computing the activation schedule with small problem sizes, we were able to verify that the solution found by this algorithm is the actual optimum.

The simulations in Section 5.1 and Section 5.3 were done with just a single or few sensor nodes. Thus they could be easily simulated with Avrora. However, if a greater number of nodes is run for a long lifetime, the performance of Avrora is not sufficient. Therefore, for the simulation of distributed assignments, which need more nodes to get meaningful results, we used a custom simulator that does not completely model the actual Mica2 hardware. Instead, it just provides the functionality to solve the problems of Section 3. Furthermore, it does not recompute a solution to the local optimization problem but uses a precomputed energy level assignment. Therefore, it cannot reflect the behavior when *Levels* switches back to a higher level near the end of its lifetime, for example. However, for comparing different algorithms that all operate on the same local optimization results, such functionality does not seem to be needed.

For these experiments we set the network lifetime to 20 time intervals. In a real application each of these intervals may include several local optimization rounds that try to maximize the average utility of the node. The required lifetime of each node was set to 8 time intervals. Half of the nodes stayed for 4 time intervals in an energy level with a utility of 2 whereas the other half stayed in this level for just 2



(a) Deviation of the number of active nodes from R_{avg}

(b) Deviation of the utility from U_{avg}

Fig. 12. Average deviation (including 95 % confidence intervals)

intervals. For the rest of their lifetime all nodes switched to a level with a utility of 0. In our experiments we placed the nodes randomly in a fixed area and varied the total number of nodes. This way we ran simulations for different network densities.

In Fig. 12(a) we show Δ_{active}, the deviation of the number of active nodes from the average. The deviation of our original approach that activates all nodes first is – as expected – the largest one since all nodes are scheduled to be active in the same time intervals. The deviations increase with greater node densities since with more nodes in the neighborhood the absolute difference of the number of active nodes also grows.

For the randomization approach the results are significantly better. In addition, if the number of nodes is increased, the average deviation only increases slightly. The reason for this reduced increase is that with more nodes it becomes more likely that the random schedules are assigned in a way that in each time interval approximately the average number of nodes is active. Therefore, the actual number of active nodes is close to the average.

Although the results of the randomization scheme are already better than those of the original algorithm, they can still be improved if some coordination among nodes is possible. These approaches consider the specific neighborhood – which is also used for computing the average deviation in the figure – instead of the complete network. As the figure shows, the results of the backtracking algorithm and of the greedy approach are almost equal. This is as expected because they both compute a solution subject to the same constraints. Small differences can be attributed to suboptimal local assignments since each node can only adjust its own schedule. Like for the random approach, the results for these two algorithms increase only slightly with higher node densities.

For the simulated annealing approach, the deviation remains constant even for higher node densities since the globally optimal solution is approximated. For the other algorithms, however, the schedules of the neighbors are fixed constraints, and only the schedule of the local node can be modified. If the node density increases, the weight of a single node decreases and more nodes are needed to balance the assignments and get closer to the average. Since the neighborhoods of the nodes are not identical, the deviation grows slightly with higher node densities for both

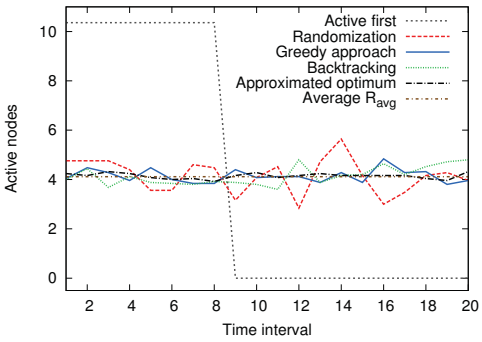


Fig. 13. Active nodes over time for a sample run of 25 nodes

the greedy and the backtracking approach.

Fig. 12(b) shows the results for balancing energy levels. Here the values do not increase with higher node densities because this figure shows the deviation from the average utility, which is divided by the number of active nodes, instead of the absolute difference in the number of active nodes. Because the time spent in the energy levels varies and because of the nodes' placement in the topology there are some small variations. This is directly reflected by the results of our original approach, where all nodes start in their highest energy level. Since with this approach nodes do not take into account the energy level assignment of their neighbors, the deviation from the average utility is almost constant.

In contrast, the deviation of the other approaches decreases if more nodes are present. The reason for this is that with more neighbors these approaches are able to approximate the average more closely. As the figure shows, the coordinating approaches are in almost all cases better than the randomization scheme. Only in very sparse topologies with less than two concurrently active nodes in the neighborhood (simulations for 5 and 10 nodes in total) the random approach might be slightly better. Since the coordination approaches operate only in the local neighborhood, they cannot balance the assignments for such a small number of neighbors. The randomization approach, in contrast, profits from the fact that all nodes throughout the network assign their schedules randomly.

Again the greedy approach is virtually equivalent to the backtracking algorithm, as their overlapping confidence intervals show. This is notable since the latter algorithm solves – unlike the former one – both sub-problems together and adjusts the activation schedule if this reduces the deviation for the utility of energy levels.

Here, the solution of simulated annealing is again better than the other approaches. This is not surprising since this approach approximates the network-wide optimum rather than solving the problem just with local knowledge on each node. Nevertheless, as the results show, the distributed approaches are very close to the optimum. However, it should be mentioned that the approximated optimum just considers a single solution for the activation schedules. Therefore, slightly better results might be possible here if the assignments of both the activation schedules and the energy levels are solved together.

Fig. 13 shows how many nodes are active in each time interval for a sample run

Table I. Effect of runtime overhead for local optimizations on node lifetime

Application	**Lifetime with *Levels***	**Reduction**
Voltage Level l_0	7.687 days	0.0 %
Voltage Level l_1	6.648 days	0.0 %
Voltage Level l_2	5.932 days	0.0 %
FFT Level l_0	944.4 days	1.8 %
FFT Level l_1	372.7 days	0.7 %

of 25 nodes. Naturally, the approach that activates all nodes first differs most from R_{avg} whereas all other approaches oscillate around this average. These oscillations are slightly greater for the randomization scheme, which leads to the greater overall deviation. The approximated optimum with simulated annealing, however, shows – as expected – the smallest deviation.

5.3 Runtime Overhead

This section evaluates the runtime overhead of *Levels*. First, we show that the overhead for the purely local optimization, where there is no coordination among nodes, is less than 2 % even for low-power applications. Second, we demonstrate that adding distributed coordination increases the overhead by just a few Joule over the lifetime of a node.

5.3.1 *Local Optimization.* Since with the purely local version of *Levels* each node determines its energy level independently from other ones, it does not have to send any radio messages. This makes *Levels* usable with low-power applications. Therefore, the only increase in energy consumption can be attributed to computational overhead. There are three sources for this overhead. First, whenever a code block belonging to an energy level is about to be executed, the system has to check if the level is active and has to add the block's energy consumption to its internal variables. Second, it accumulates the continuously consumed energy every few seconds. Finally, every few hours, *Levels* tries to adjust the current energy level with the Simplex algorithm.

To find out the actual effects of the computational overhead on node lifetime, we simulated Voltage (see Section 5.1) and FFT, a low-power application that repeatedly computes a Fast Fourier Transform in an energy level, with and without our runtime system doing its computations. As the results in Table I show, for short-lived applications with a lifetime of only a few days, the energy overhead of the computation does not result in a detectable decrease in node lifetime. Even for extremely low-power applications with a lifetime of several years, the CPU overhead of our runtime system leads to a reduction in lifetime of less than 2 %. For FFT this corresponds to an increase in average power consumption of just 6.9 μW. Compared to an overall power consumption of 385 μW in Level l_0 and 987 μW in Level l_1 this increase is negligible.

5.3.2 *Distributed Assignment.* The overhead for coordinating assignments is also very small, especially since this optimization is run less frequently and requires only minimal input from directly neighboring nodes. Table II summarizes the main components of the overhead for a typical low-power application. Again the numbers have been obtained using Avrora.

Table II. Energy overhead of distributed level assignments

Action	**Energy**
Send packet with LPL	70.9 mJ
Receive for five minutes with LPL	156 mJ
Greedy algorithm	0.199 mJ

The table assumes that the radio has to be turned on for five minutes just to exchange the schedules. During this time, we make use of low-power listening (with 1 % duty cycle) that increases the length of the preamble in order to reduce the time receivers have to listen for radio messages [Polastre et al. 2004].

If 20 time intervals are used, a node's state can be transmitted in a single network packet. In applications with up to 128 energy levels (much more than expected in real-world applications) for each time interval, just a single byte is needed: 1 bit to represent the on/off state and up to 7 bits for the energy level assigned to the time interval. Sending such a message including the longer preamble for low-power listening consumes on the Mica2 platform 70.9 mJ.

Furthermore, nodes have to be able to receive network packets for coordination. In loosely synchronized networks, it should be sufficient to turn on the radio for a few minutes each time when a node expects to receive new schedules from its neighbors. Turning on the radio for five minutes with low-power listening and the accompanying switch of the CPU out of the power-save mode requires 156 mJ.

Finally, the last component of the overhead is computing the local schedule using our greedy algorithm. For 20 time intervals, this computation takes 9.39 ms and requires just 0.199 mJ of energy.

In summary, the energy needed for communication dominates the overhead for computation. One coordination cycle requires 227 mJ in total. For an initialization at the beginning and 20 computation cycles during the lifetime of the network the coordination consumes just 4.77 J. Compared to a total battery capacity of about 32,000 J this is negligible. Even if each message was sent several times to increase reliability, an effect on node lifetime would hardly be detectable.

6. RELATED WORK

In this section, we give a brief overview of work related to *Levels*. Particularly, we describe systems that take into account energy considerations for adaptation, extend network lifetime by deactivating redundant nodes, map energy consumption to code blocks, and model battery behavior.

In the realm of mobile computing, Odyssey [Flinn and Satyanarayanan 1999] monitors the available energy and adapts the fidelity of applications to meet a user-defined lifetime goal. For example, a video player switches to a differently compressed source file or reduces its window size if energy becomes scarce. Odyssey does not provide a programming abstraction like our energy levels and does not leverage simulation data. Furthermore, it has been designed for less resource-constrained devices and relies on highly accurate measurement equipment, which we cannot assume on inexpensive sensor nodes.

Similarly, ECOSystem [Zeng et al. 2002] tries to achieve a target lifetime by limiting the discharge rate of the battery. It introduces the Currentcy Model to deal

with the demands of competing tasks in a multitasking system. Rather than identifying optional functionality in applications, it modifies the scheduler to execute only those tasks that have not spent their energy budget for the current round yet. Unlike our approach, it does not exploit information from simulation and, therefore, has to do detailed energy accounting at runtime.

As already mentioned in Section 2, Eon, which has been developed concurrently to our work, provides another language and runtime environment for energy-aware sensor network applications. However, instead of dealing with lifetime goals, Eon focuses on perpetual systems that employ energy harvesting. Therefore, it pursues the goal of balancing energy consumption and production. Eon does not make use of information from simulation and, therefore, can only estimate the energy consumption of different functionalities. In addition, it does not balance the functionalities of different nodes.

TinyDB [Madden et al. 2005] allows to adapt the interval between the measurements of a query in order to meet user-defined lifetime goals. Similar to our rationale, its authors argue that in environmental monitoring scientists are more concerned about meeting a lifetime goal than about the sampling rate. Since TinyDB's programming interface is based on high-level SQL-like queries, changing the sampling rate is the only way to influence network lifetime.

There is already a large body of work dealing with the coverage problem in wireless sensor networks. This work switches redundant nodes into sleep mode to maximize the time that a given area is monitored by the network [Cardei and Wu 2006; Huang et al. 2006]. Closely related are topology control mechanisms like ASCENT [Cerpa and Estrin 2002] that switch off redundant nodes but strive to preserve network connectivity. PEAS [Ye et al. 2003] controls the network density to ensure both coverage and connectivity. Likewise, duty-cycling approaches [Giusti et al. 2007] periodically turn off nodes to extend network lifetime. Unlike *Levels*, all of these approaches are only targeted to dense networks where redundant nodes can be temporarily deactivated. In addition, although they keep the application quality roughly constant, they do not have any given lifetime goals but try to maximize the time for which they provide coverage or connectivity, respectively. Finally, they do not provide more states with differing functionality – like our energy levels – and usually turn nodes just on or off.

Sensor network simulators like Avrora [Titzer et al. 2005; Landsiedel et al. 2005] and PowerTOSSIM [Shnayder et al. 2004] enable the prediction of the energy consumption of a sensor node. The values obtained from these tools are often used for evaluation purposes and to give the developer hints about energy consumption, although usually not at runtime. Avrora allows to break down energy consumption to individual functions. However, this part of Avrora can only associate the energy consumption of the CPU with some code rather than including the other hardware components on a node. In addition, unlike our approach, it does not take into account the energy consumed by functions that are called by the code under measurement or, in the case of TinyOS, by asynchronously executed tasks.

There are several more advanced battery models than ours described in the literature [Rao et al. 2003]. They take into account effects resulting from temperature changes and time-varying loads, for example. However, because the voltage sensor

on typical sensor nodes cannot provide the precision of lab equipment, we have to deal with inaccuracies anyway. Furthermore, the computational overhead of many accurate battery models is too large for resource-constrained sensor nodes, and it takes even for more powerful computers hours to simulate a load profile.

7. CONCLUSIONS

In this paper, we have described and evaluated *Levels*. Unlike existing approaches that try to maximize network lifetime, *Levels* provides mechanisms to meet user-defined lifetime goals for a sensor network and for each of its individual nodes. Since in applications like structural health monitoring and environmental monitoring the user knows the required lifetime in advance, this approach helps to maximize application quality for that time.

Levels requires only small modifications to existing code, and its energy levels offer a flexible and easy-to-use programming abstraction. *Levels* allows programmers to mark code that is not needed to provide some basic functionality like network connectivity or sampling with less energy-intensive sensors. In addition, estimating the energy consumption of parts of an application is easy with our simulation-based approach for energy profiling. Finally, our runtime system shields the application developer completely from low-level issues related to lifetime estimation. Even balancing energy levels and node schedules among neighboring nodes for constant application quality can be performed by the system without any effort for the developer.

If an accurate battery model and information about the energy consumption of the sensor nodes are available, *Levels* helps to ensure that each node meets its lifetime goal and provides an application quality that is close to the optimum. Furthermore, even with the very simple battery model used in our experiments, the sensor nodes achieved their target lifetime in almost all cases.

Using our coordination algorithm, nodes can balance their energy level assignments and, in dense networks with redundant nodes, their activation schedules in order to provide almost constant application quality. This approach is run in a completely distributed way and requires only minimal data exchange among nodes. As we have shown in the evaluation, the energy overhead of both this coordination and the local optimization is negligible.

In conclusion, we expect that *Levels* will help to make the creation of adaptive and energy-aware sensor network applications much easier. Although switching to lower energy levels might somewhat decrease the quality of the data obtained from the network, we argue that a node is more useful when providing reduced functionality than if it stops working completely. Furthermore, with our coordination approach, *Levels* increases the probability that there are always some nodes providing a high level of functionality.

Regarding future work, we plan to adapt *Levels* to applications that want to achieve a given quality while maximizing their lifetime. Similarly, the effects of energy harvesting – with the possibility of perpetual applications – should be studied separately in more detail. Finally, adjusting the time interval between computations dynamically could possibly further improve application quality.

REFERENCES

Camacho, E. F. and Bordons, C. 2004. *Model Predictive Control*, 2nd ed. Advanced Textbooks in Control and Signal Processing. Springer-Verlag.

Cardei, M. and Wu, J. 2006. Energy-efficient coverage problems in wireless ad-hoc sensor networks. *Computer Communications 29,* 4, 413–420.

Cerpa, A. and Estrin, D. 2002. ASCENT: Adaptive self-configuring sensor networks topologies. In *Proc. of the Twenty-First Annual Joint Conf. of the IEEE Computer and Communications Societies*. Vol. 3. 1278–1287.

Chvátal, V. 1983. *Linear Programming*. W. H. Freeman and Company.

Dunkels, A., Österlind, F., Tsiftes, N., and He, Z. 2007. Software-based on-line energy estimation for sensor nodes. In *Proc. of the Fourth Workshop on Embedded Networked Sensors*.

Duracell Batteries. 2001. Duracell Plus alkaline-manganese dioxide battery. `http://www.mdsbattery.co.uk/datasheets/duracell/MN1500PL.pdf`.

Dutta, P., Feldmeier, M., Paradiso, J., and Culler, D. 2008. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *Proc. of the International Conference on Information Processing in Sensor Networks*. 283–294.

Flinn, J. and Satyanarayanan, M. 1999. Energy-aware adaptation for mobile applications. In *Proc. of the Seventeenth ACM Symposium on Operating Systems Principles*. 48–63.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the Conf. on Programming Language Design and Implementation*. 1–11.

Giusti, A., Murphy, A. L., and Picco, G. P. 2007. Decentralized scattering of wake-up times in wireless sensor networks. In *Proc. of the 4th European Conference on Wireless Sensor Networks*. 245–260.

Huang, C.-F., Lo, L.-C., Tseng, Y.-C., and Chen, W.-T. 2006. Decentralized energy-conserving and coverage-preserving protocols for wireless sensor networks. *ACM Trans. Sen. Netw. 2,* 2, 182–187.

Jiang, X., Dutta, P., Culler, D., and Stoica, I. 2007. Micro power meter for energy monitoring of wireless sensor networks at scale. In *Proc. of the 6th Int'l Conf. on Information Processing in Sensor Networks: Track on Sensor Platforms, Tools and Design Methods*. 186–195.

Kim, S., Pakzad, S., Culler, D., Demmel, J., Fenves, G., Glaser, S., and Turon, M. 2007. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. of the 6th International Conference on Information Processing in Sensor Networks*. 254–263.

Kirkpatrick, S., Gelatt Jr., C. D., and Vecchi, M. P. 1983. Optimization by simulated annealing. *Science 220,* 4598, 671–680.

Lachenmann, A., Marrón, P. J., Minder, D., and Rothermel, K. 2007. Meeting lifetime goals with energy levels. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems*. 131–144.

Landsiedel, O., Wehrle, K., and Götz, S. 2005. Accurate prediction of power consumption in sensor networks. In *Proc. of the Second Workshop on Embedded Networked Sensors*.

Liu, T., Sadler, C. M., Zhang, P., and Martonosi, M. 2004. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proc. of the International Conference on Mobile Systems, Applications, and Services*. 256–269.

Madden, S. R., Franklin, M. J., Hellerstein, J. M., and Hong, W. 2005. TinyDB: An acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30,* 1, 122–173.

Marrón, P. J., Saukh, O., Krüger, M., and Grosse, C. 2005. Sensor network issues in the Sustainable Bridges project. In *European Projects Session of EWSN 2005*.

Polastre, J., Hill, J., and Culler, D. 2004. Versatile low power media access for wireless sensor networks. In *Proc. of the Int'l Conf. on Embedded Networked Sensor Systems*. 95–107.

Polastre, J., Szewczyk, R., and Culler, D. 2005. Telos: Enabling ultra-low power wireless research. In *Proc. of the Int'l Conf. on Information Processing in Sensor Networks: Special Track on Platform Tools and Design Methods for Network Embedded Sensors*.

Rao, R., Vrudhula, S., and Rakhmatov, D. N. 2003. Battery modeling for energy-aware system design. *Computer 36,* 12, 77–87.

Shnayder, V., Hempstead, M., Chen, B.-r., Werner-Allen, G., and Welsh, M. 2004. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the 2nd Int'l Conf. on Embedded Networked Sensor Systems*. 188–200.

Simon, G., Maróti, M., Ákos Lédeczi, Balogh, G., Kusy, B., Nádas, A., Pap, G., Sallai, J., and Frampton, K. 2004. Sensor network-based countersniper system. In *Proc. of the 2nd International Conference on Embedded Networked Sensor Systems*. 1–12.

Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. 2007. Eon: A language and runtime system for perpetual systems. In *Proc. of the 5th International Conference on Embedded Networked Sensor Systems*. 161–174.

Titzer, B., Lee, D., and Palsberg, J. 2005. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of the Fourth Int'l Conf. on Information Processing in Sensor Networks*. 477–482.

Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., Burgess, S., Dawson, T., Buonadonna, P., Gay, D., and Hong, W. 2005. A macroscope in the redwoods. In *Proc. of the 3rd International Conference on Embedded Networked Sensor Systems*. 51–63.

Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., and Welsh, M. 2006. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of the Symp. on Operating Systems Design and Implementation*.

Ye, F., Zhong, G., Cheng, J., Lu, S., and Zhang, L. 2003. PEAS: A robust energy conserving protocol for long-lived sensor networks. In *Proc. of the Int'l Conf. on Distributed Computing Systems*. 28–37.

Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. 2002. ECOSystem: Managing energy as a first class operating system resource. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*. 123–132.