Flexible Online Energy Accounting in TinyOS

Simon Kellner

System Architecture Group
Karlsruhe Institute of Technology
kellner@kit.edu

Abstract. Energy is the most limiting resource in sensor networks. This is particularly true for dynamic sensor networks in which the sensor-net application changes its hardware utilization over time. In such networks, offline estimation of energy consumption can not take into account all changes to the application's hardware utilization profile and thus invariably returns inaccurate estimates. Online accounting methods offer more precise energy consumption estimates. In this paper we describe an online energy accounting system for TinyOS consisting of two components: An energy-estimation system to collect information about energy consumption of a node and an energy-container system that allows an application to collect energy-consumption information about its tasks individually. The evaluation with TinyDB shows that it is both accurate and efficient.

Keywords: energy accounting policy tinyos.

1 Introduction

Energy still is the most critical resource in sensor networks. Limitations on energy supply as well as on other resources have led to operating system designs that offer only minimalistic hardware abstractions. The core of TinyOS, for example, is an event-based system that helps application developers in dealing with asynchronous hardware requests, and little else. One effect of this design decision is to make developers more considerate about hardware usage and therefore energy consumption. TinyOS makes it hard to actively wait for a hardware event to occur, while making it easy to react to the same event, which is the more energy-efficient approach in most situations.

One approach to designing sensor-net applications that meet pre-defined energy consumption requirements is to develop an application whose hardware utilization pattern is simple enough to allow predictions on the application's energy consumption. Global parameters of such applications can then be changed to accommodate energy consumption requirements. But the lack of convenient hardware abstractions does not necessarily limit developers in creating complex applications. A sensor network running the TinyOS-based TinyDB application, for example, allows users to issue (SQL-like) queries to the sensor network at a time of their choosing. Planning the energy consumption of nodes in this network can not be done a-priori, because the energy consumption characteristics of a node running TinyDB change with the queries it processes.

P.J. Marron et al. (Eds.): REALWSN 2010, LNCS 6511, pp. 62–73, 2010.

Control of energy consumption in this scenario is only feasible using online energy accounting on the sensor nodes. Information on the energy consumption of whole nodes, however, does not offer much information. An energy-intensive query might, for example, be only revealed by comparing node energy consumption before and after a query was sent into the network. Energy consumption of queries, on the other hand, can be readily used to decide if a query consumes too much energy and has to be canceled before it wears down the energy supplies of the sensor network.

This paper makes the following contributions:

- An online energy-estimation system for TinyOS that allows sensor nodes to become aware of their energy consumption.
- An energy container system for TinyOS that allows application developers to collect energy-consumption information about control flows in the application.
- A set of accounting policies that can be used to adapt the energy-container system to its purpose as set by the application developer.

The paper is structured as follows: After presenting related work in Sect. 2 we define a usage scenario in Sect. 3 that will be referenced later on. Then we present the design and selected implementation issues of the energy estimation system (Section 4) and the energy container system (Section 5). Section 6 details several accounting policies of our energy-container system. Following an evaluation of our systems in Sect. 7 we conclude with an outline of future work in Sect. 8 and closing remarks in Sect. 9.

2 Related Work

Management of energy in sensor networks has received significant attention in research over the last years, as it concerns the primary resource of such networks.

PowerTOSSIM [7] is similar to our own energy estimation system. It instruments OS components or simulations thereof to track power states and uses an energy model to compute energy consumption for one or more sensor nodes. PowerTOSSIM, however, targets off-line simulation, whereas our instrumentation and model are designed to be used in on-line energy accounting.

AEON [5] is the energy model used in the Avrora [8] simulator. It models the hardware's power states of a MICA2 node. Our energy model is based primarily on the MICAz node and additionally considers transitions between hardware states.

Schmidt, Krämer et al. [6] present another energy model used to make existing simulators energy-aware. Although they mention the potential to use their energy model in online energy estimation, they do not elaborate on that option further.

Dunkels et al. [2] present an energy-estimation system for the Contiki OS. This system is used to estimate energy consumption per hardware components. We employ a similar energy-estimation system and extend it with energy containers to a full energy-accounting system that is able to account energy based on control flows, which may span multiple hardware components.

Quanto [3] is an energy profiling mechanism for TinyOS that accounts energy consumption information of activities in an application. It employs a hardware energy meter to measure the total energy consumption of a sensor node, and tries to break this information down to energy consumption of individual hardware components. Our energy estimation system does not require any hardware instrumentation. It also provides more options for accounting policies, facilitating more use cases than energy profiling.

Resource Containers [1] are an abstraction in an operating system (OS) introduced by Banga, Druschel and Mogul providing flexible, fine-grained resource accounting on web-servers. The main idea is to separate execution (processes) from resource accounting, so that an application itself can define the entity being subject to accounting. In operating systems featuring CPU abstractions such as threads or processes, Resource Containers give administrators and users the ability to account all activity connected to a user request, which usually has a higher significance than process-based accounting. We adapt this concept to TinyOS and focus solely on energy as a resource. Consequently, we call our containers *energy containers*.

3 Scenario

In this paper we use the following reference scenario. A network of sensors is programmed with a dynamic application like TinyDB. We assume multiple users of the sensor network who periodically retrieve data from the network. They retrieve data by injecting queries into the network, which are then periodically processed by the application on the sensor nodes until a user stops them. The sensor-net application can process multiple queries concurrently over a long period of time.

Network operators and users should be able to intervene in the query processing to save energy.

4 Online Energy Estimation

An important part of the energy accounting system is the on-line estimation of a sensor node's energy consumption.

We recognize a sensor node as a collection of simple, independent hardware components controlled by one microcontroller (MCU). Therefore, we model a node's energy consumption using a collection of small state machines, one for each independent hardware component.

These state machines have a state for each distinguishable hardware power state, i.e., a hardware state exhibiting a characteristic current draw. Each state s is annotated with this current draw I_s. Transitions t in this state machine are annotated with the amount of electric charge Q_t they consume.

To compute an estimation of the energy a hardware component has consumed, we record the time T_s the hardware spent in each state s as well as the number of times N_t each transition t occurred and compute the estimated consumed energy E as

$$\frac{E}{V} = \sum_s T_s I_s + \sum_t Q_t N_t \ . \tag{1}$$

This is the same idea Dunkels et al. [2] use in their online energy estimation for the Contiki operating system.

We implemented the online energy estimation method presented above in TinyOS 2 for the MICAz platform. At the time of writing, we have energy estimation implementations for the ATmega128 microcontroller, the CC2420 radio chip, the LEDs, and the magnetometer on the MTS300 sensor board.

5 Energy Container System

To store the estimated energy usage per query, we employ a hierarchy of energy containers. With multiple energy containers in the system (e.g., for concurrent queries), we need some help from the application (e.g., TinyDB) to map activities to energy containers. Our energy-container system keeps this association intact.

In this section we will first present energy-container types and their structure in our system. We then will describe the way in which the application should interact with the energy container system. Afterwards we will detail how our system keeps energy-container associations intact.

5.1 Energy Container Structure

In our system, energy containers are hierarchically structured, as shown in Fig. 1.

Most containers in our energy container system are under the control of the application developer. They can be allocated, read, and released at the application's discretion. The application can also switch between containers, indicating that subsequent computations should be accounted to a different container.

In addition to these *normal containers*, the *root container* holds the energy consumption of a whole sensor node: All energy consumption is accounted both to the container selected by the application and to the root container. Together, these two types of containers form a flat hierarchy.

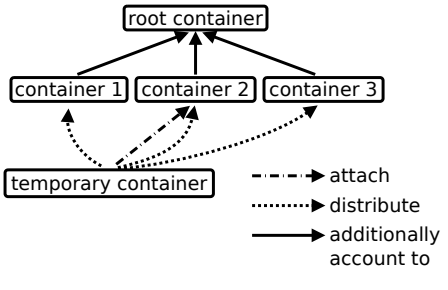

Fig. 1. Hierarchy of energy containers

```
ec_id newContainer();
void attachToContainer(ec_id id);
void switchToContainer(ec_id id);
uint32_t getContainer(ec_id id);
uint32_t getRootContainer();
void stopMonitoring(ec_id id);
```

Fig. 2. Interface to the container system

As a special case, a *temporary container* is used in cases where the application cannot know (yet) to which container the current energy consumption should be accounted. On a sensor node such a situation only occurs whenever a message is received by the radio: The message can belong to the query currently active, lead to the creation of a new query, or it could not be associated to query processing at all. For these cases we use one extra container that is activated upon reception of a message and is treated in a special way: When the message is found to belong to a known query, or creates a new one, the application has to *attach* the temporary container to the normal container used for that query. In this case the contents of the temporary container are added to the normal container, the currently active container is set to the normal container, and the temporary container is deactivated. If the application does not attach the temporary container to a normal one, our container system apportions the energy accounted to the temporary container among all currently active containers at the end of the message reception handler routine.

5.2 Energy Container Interface

Figure 2 shows the interface an application should use to work with our energy container system. The commands are ordered as they would be used in an application.

Upon reception of a query message, the application attaches itself to a known container (`attachToContainer`) or to a newly created one (`newContainer`). Before an application starts a processing step of a query, it switches to the container created for that query (`switchToContainer`). When creating a query response message, the application may choose to include the contents of one or more energy containers (`get{,Root}Container`). If a query should no longer be processed (i.e., removed from the system), it invokes `stopMonitoring` to completely deactivate the indicated container.

The presented interface intentionally does not provide full control over energy containers. For instance, there is no command to deactivate the currently active container without either removing it completely from the system or switching to another container. In our opinion, every action on a sensor node should be accountable to a request made by a user of a sensor network. Nevertheless it is possible to create new containers, for example, to account a maintenance task that is run on a sensor node and is independent of any queries.

Also intentionally absent from the energy container interface are commands to operate on the contents of the energy containers. We separate operations on energy containers within our energy container system from the operations on energy values in the application. It is the responsibility of the application to make network-wide use of these locally obtained energy values. Our energy-container system can not automatically handle all cases of aggregation that an application may perform on energy values.

5.3 Energy Container Implementation

As the previous section on the energy container interface shows, the application has to be modified to use energy containers. Naturally we strive to require as few changes to the application as possible, which means that our system has to keep associations to energy containers intact.

Resource containers in UNIX are attached to threads, so thread control blocks would be used to store references to associated resource containers, and the reference to the currently active thread would be used to access the currently active resource container. TinyOS, however, does not provide a CPU abstraction such as threads. TOSthreads, a TinyOS library providing threads, is optional and many applications do not require it, including TinyDB. TinyOS at its core therefore lacks a structure like a thread control block and a reference to the currently active thread.

In the absence of threads, we have to implement container associations in a different way: We use a set of TinyOS components to track the control flow of an application and to keep an energy container associated to this control flow. By *control flow* we mean a series of actions (instruction execution, hardware operation) where every action is a direct consequence of the actions preceding it. For example, a control flow to sample a value from a sensor can comprise: issuing a `read()` call, turning on the sensor, configuring it, reading it, turning the sensor off, and returning the value to the application in a `readDone` event.

A control flow can be suspended several times during its course through TinyOS, but all these suspensions stem from two cases: software queues and hardware operations. We say that a control flow is suspended when a piece of code performs an enqueue operation, and that it is resumed on the related dequeue operation. Similarly, we say that a control flow is suspended when it starts a hardware operation, and that it is resumed when the hardware causes an interrupt handler to be executed.

Software queues are frequently used in TinyOS. For example, instances of software queues are the queue of timer events in virtual timers, message output queues of communication modules, access request queues of shared resources, and the scheduler queue.

We instrumented several TinyOS components to send information about enqueue-, dequeue-, and hardware operations to our subsystem for control flow tracking. For software queues we implemented shadow queues of energy containers. During the enqueue operation, the shadow queue enqueues a reference to the currently active energy container, thereby associating the object being enqueued with this energy container. When an object is dequeued, the corresponding energy container from the shadow queue is activated, resuming the control flow with the energy container association intact. Control flow tracking over software queues is illustrated in Fig. 3.

Figure 4 shows that control flow tracking in hardware is handled differently. Not only is there just one energy container association that has to be stored, but, more importantly, the concurrency of hardware operations and program execution on the microcontroller means that there can be multiple active energy

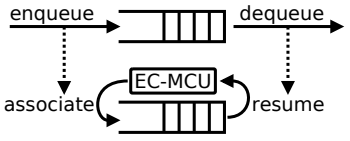

Fig. 3. Control flow tracking of software queues

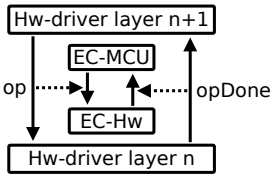

Fig. 4. Control flow tracking of hardware operations

containers on one node, each one associated with a different hardware component. Thus, our control-flow subsystem copies the current energy container association from the microcontroller to the hardware, and copies the association back when a hardware signal is received.

Altogether, the control flow tracking components ensure that an application has to switch containers only inside its own components, and only on switching query processing from one query to another.

6 Accounting Policy

Energy consumed during use of a hardware component is accounted to the energy container associated with the active control flow on the hardware. But energy is also consumed by the hardware before and after use: On startup, on shutdown, and between uses. We call this kind of energy consumption *collateral*.

There are many ways in which collaterally consumed energy can be accounted. The choice between these ways depends on the hardware usage pattern of the operating system, as the usage pattern defines whether collaterally consumed energy can be shared or not. It also depends on the reason why energy accounting is used: For energy profiling purposes, for example, an application developer might prefer not to account collaterally consumed energy at all, or account it to a separate container. A provider of a TinyDB network, however, might prefer to have all energy consumption accounted to TinyDB queries in a fair manner. As energy profiling systems already exist, we focus on a fair energy accounting in the TinyDB scenario. We identify hardware usage patterns and choose a suitable apportioning policy.

6.1 Single Use

The hardware usage pattern of the microcontroller (MCU) is simple: Its startup overhead is negligible, and then there is only one active control flow at a time. The apportioning policy used on energy consumed by the MCU is equally simple: Account energy consumption to the active energy container, or, if there is no active energy container, distribute it evenly among all normal energy containers in the system.

6.2 Shared Use

If the startup overhead of a hardware component is not negligible, other apportioning policies must be used. A policy suited for most devices is to share the collaterally consumed energy among all containers which were associated in the time interval between startup and shutdown of a hardware component. The collaterally consumed energy could either be apportioned evenly to these containers, or proportionally to their hardware usage. We use an evenly apportioning policy for the magnetometer sensor on the MTS300 sensor board, which has a large startup overhead (waiting 100 ms for the sensor to stabilize) and negligible use costs (taking an A/D converter sample is done in a few clock cycles of the MCU). The implementation is integrated into the ICEM [4] framework for shared devices in TinyOS, so that other devices may easily be instrumented as well.

We use the same policy to account the energy consumption of the radio chip in the "low-power listening" mode offered by TinyOS. In this mode, TinyOS repeats the transmission over a configurable time interval, until it either receives an acknowledgment, or a timeout occurs. A node that should receive messages can thus settle on periodically checking for transmissions and keeping the radio chip turned off between checks. The repeated attempts at sending a message can be viewed as a form of synchronization: Barring radio noise, if more messages are sent to the same receiver immediately after one transmission attempt succeeded, those messages will arrive on their first transmission attempt. We treat all transmission attempts but one (the successful one) as synchronization overhead to be accounted to all energy containers of successive messages to the same receiver.

6.3 Continuous Use

Yet another different policy is needed for the radio chip if the application is not configured to use energy-saving mechanisms such as low-power listening. In this case, TinyOS keeps the radio powered on continuously. The absence of use intervals makes it difficult to assign a fair share of collaterally consumed energy to a container. We employ a log of all energy containers that were used to send or receive messages, and apportion collaterally consumed energy of the radio chip to all these containers using a geometric distribution, so that containers using the radio more often will bear most of the energy consumption.

7 Evaluation

We evaluated our energy container system using TinyDB. As a first step, we ported TinyDB to TinyOS 2.1.0. TinyDB is a large sensor-net application consisting of over 140 files with a total of over 25,000 lines. It does not fit in the program memory of a TelosB node (48 kBytes) and uses nearly all program memory of a MICAz node ($\sim$ 60 of 64 kBytes), even with several features such as query sharing and "fancy" aggregations deactivated. The output file of the nesC

compiler comprises nearly 40,000 lines of code when TinyDB is compiled for MICAz nodes.

TinyDB is a dynamic sensor-net application in that it allows users to inject queries at run-time, and allows to run a limited number of different queries simultaneously. This makes it an ideal application to benefit from our flexible online energy accounting system.

We evaluated our system with regard to the following aspects:

- Ease of use: The work required to add energy containers to TinyDB.
- Overhead: The additional costs of using energy containers.
- Accounting fairness: Fairness of energy consumption distribution.
- Accuracy: Accuracy of the energy estimation system.

7.1 Experimental Setup

In our evaluation we used two TinyDB applications: *TinyDB-noec* is a regular TinyDB application.

In *TinyDB-full,* which is based on TinyDB-noec, we create an energy container for each new query, and send the energy consumption information in this container back to the base station.

To measure the estimation error of our energy estimation system, we additionally modified TinyDB-full to include a new field in status messages. In this field TinyDB-full reports the difference of the current root container contents to its contents when the first query injection message arrived. Immediately after terminating the last active query on our measured sensor node, we sent a status request message and recorded the energy reported in the status message. Differences between the reported energy values and the measured energy consumption are caused by errors in the energy estimation system.

We used three queries that exhibit different hardware usage. Each of these queries is periodically processed by TinyDB in so-called *epochs*, each epoch being about 750 ms in length by default. At the begin of an epoch, result values are computed for each query, and at the beginning of the next epoch, they are sent out in a query result message. The queries run until they are stopped by a user.

One query, `select nodeid, qids`, uses only information already present in the microcontroller, namely the ID of the node and the IDs of the currently active queries. We used two versions of this query, one using default settings (`sample period 1024`) and one having an epoch length of double the default value (`sample period 2048`).

The third query used, `select nodeid, mag_x`, samples the x-direction of the magnetometer on a MTS300 board, which makes this query consume significantly more energy than the first one.

7.2 Ease of Use

To provide energy containers in TinyDB-full, we had to add 59 lines of code and to make small changes to 5 lines of code. About half of these changes were straightforward changes, like adding fields to message structures and filling them.

7.3 Overhead

We measured two kinds of overhead in our test application: One is the increased code size and memory usage, the other one is additional energy consumption.

As Table 1 shows, adding energy containers to TinyDB caused close to 4000 lines of code to be included in the C file generated by the nesC compiler (which contains the whole application).

Table 1. Sizes of the applications used in our evaluation. Lines of (C) code as reported by cloc (`cloc.sourceforge.net`), Program size and Memory usage as reported by the TinyOS build system.

Application	Lines of code	Program size [byte]	Memory usage [byte]	Avg. current draw [mA]
TinyDB-noec	39175	57382	3292	23.375
TinyDB-full	42971	63552	3449	23.312

We also measured the energy consumption overhead caused by our energy container system. To this end we ran one query (`select nodeid, mag_x`) for about 40 seconds on each of our applications multiple times and measured the current draw. The average current draw is also shown in Table 1. The difference in current draws is 63.1 µA, which is only slightly larger than the standard deviation of the average current draws (which was 31.1 µA for TinyDB-noec and 45 µA for TinyDB-full).

7.4 Accounting Fairness

As an example of how energy containers could be used, we issued two queries with different hardware usage: Both queries requests only information about the software, which is available at virtually no cost (`select nodeid, qids`), but at different sample rates. Query 2 (`sample period 1024`) should send at double the rate of Query 1 (`sample period 2048`). Query 2 is injected after Query 1 and stopped before Query 1, so that energy is accounted first to one, then two, and again one container.

When both queries are active and synchronized, the radio should be used alternately by one and two queries. We configured the sensor node to use the low-power listening mode of TinyOS, and used a shared policy to account collaterally consumed energy on the two energy containers of the queries.

The energy container contents of the queries are reported in the query result messages. Figure 5 shows these energy values plotted as they are sampled at the sensor node. Also shown in the figure is the sum of the most recent energy values of both queries, which should closely resemble the measured energy consumption.

Figure 5 shows that query 2 draws more power than query 1, which can be explained by its higher message sending rate. Query 1 profits from Query 2 in that it is charged with less energy consumption when Query 2 is active.

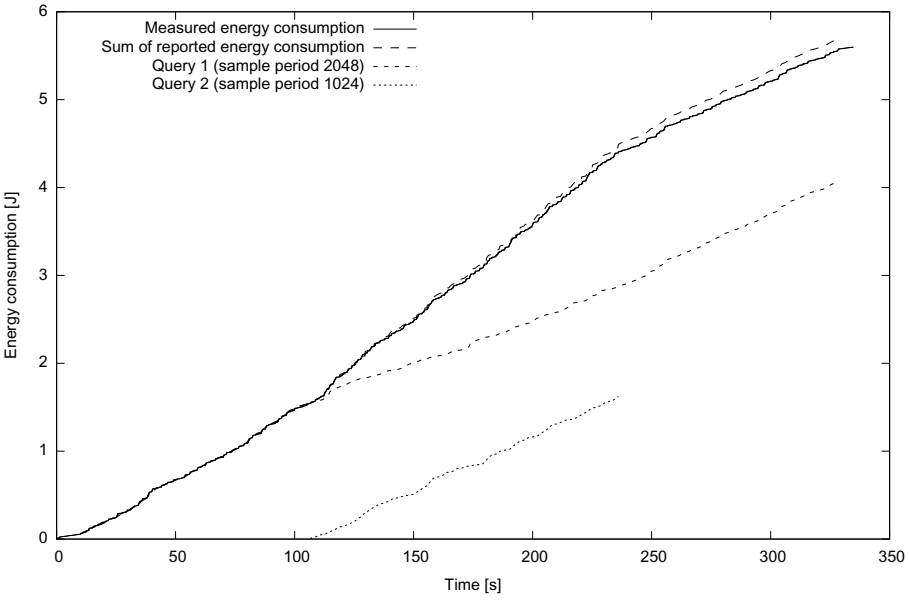

Fig. 5. Energy consumption reported by queries

7.5 Accuracy

To determine the accuracy of our energy estimation system, we measured the real energy consumption of our node and compared the measurements to the contents of the node's root energy container in all of the tests involving TinyDB-full, i.e., some of the overhead tests and the previous example.

The energy consumption recorded in the root container was within 3 % of the measured energy consumption.

8 Future Work

In further work, we plan to improve our implementation to support a greater variety of hardware. Preliminary measurements indicate that the supply voltage has an effect on current draw that varies between chips. We are looking on how best to capture this behavior appropriately in our energy model.

We also plan to incorporate distributed energy management into TinyDB that makes use of our energy-container system.

9 Conclusion

In this paper, we described a flexible online energy accounting system for TinyOS, the basis of which is an online energy estimation system. We introduced energy containers in TinyOS as specialized resource containers, allowing us to account energy consumption of parts of a sensor-net application separately. Evaluation of our implementation shows it to be accurate and to have a low energy overhead.

References

1. Banga, G., Druschel, P., Mogul, J.: Resource containers: A new facility for resource management in server systems. In: Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI 1999), pp. 45–58 (February1999), http://www.cs.rice.edu/~druschel/osdi99rc.ps.gz
2. Dunkels, A., Österlind, F., Tsiftes, N., He, Z.: Software-based on-line energy estimation for sensor nodes. In: Proceedings of the 4th workshop on Embedded networked sensors (EMNETS 2007), pp. 28–32. ACM, New York (2007)
3. Fonseca, R., Dutta, P., Levis, P., Stoica, I.: Quanto: Tracking energy in networked embedded systems. In: Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation (OSDI 2008), pp. 323–338. USENIX Association (December 2008), http://www.usenix.org/events/osdi08/tech/full_papers/fonseca/fonseca.pdf
4. Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D., Levis, P.: Integrating concurrency control and energy management in device drivers. In: Proceedings of the twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP 2007), pp. 251–264. ACM, New York (2007)
5. Landsiedel, O., Wehrle, K., Götz, S.: Accurate prediction of power consumption in sensor networks. In: Proceedings of the second IEEE Workshop on Embedded Networked Sensors (EmNetS-II), pp. 37–44 (May 2005)
6. Schmidt, D., Krämer, M., Kuhn, T., Wehn, N.: Energy modelling in sensor networks. Advances in Radio Science 5, 347–351 (2007), http://www.adv-radio-sci.net/5/347/2007/ars-5-347-2007.pdf
7. Shnayder, V., Hempstead, M., Chen, B., Werner-Allen, G., Welsh, M.: Simulating the power consumption of large-scale sensor network applications. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, SenSys 2004, pp. 188–200. ACM Press, New York (2004)
8. Titzeri, B.L., Lee, K.D., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks, IPSN 2005, p. 67. IEEE Press, Piscataway (2005)