Application-Informed Radio Duty-Cycling in a Re-Taskable Multi-User Sensing System

Omprakash Gnawali, Jongkeun Na, Ramesh Govindan
Computer Science Department, University of Southern California
gnawali@usc.edu, jkna@enl.usc.edu, ramesh@usc.edu

ABSTRACT

As sensor networks mature, there will be an increasing need for re-usable, dynamically taskable software systems that support multiple concurrent applications. In this paper, we consider the problem of energy management in such systems, taking Tenet as a case study. Our work considers energy management under three new constraints: dynamic multi-hop routing and tasking, multiple concurrent applications, and reliable end-to-end data delivery. We present AEM, an energy management system that satisfies these constraints. AEM statically analyzes and infers the traffic profile for the application and accordingly tunes the duty-cycling protocol to provide the best trade-off in latency and data delivery performance. Furthermore, unlike other duty-cycling protocols with pre-computed or fixed transmission and reception time slots, AEM uses elastic schedules that allows it to adapt to dynamics while enabling bounded latency of event detection. Our experiments show that AEM achieves 1-3% duty-cycles, while allowing concurrent applications to transmit 100% of the sensor data in a multi-hop 40-node network testbed.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*

General Terms: Design, Experimentation, Performance

Keywords: Sensor networks, energy efficiency

This material is based on work supported by the National Science Foundation under Grant No. CNS-0520235 and the Korea Research Foundation Grant funded by the Korean Government [KRF-2008-357-D00220]. Any opinions, findings and conclusions or recomendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IPSN'09, April 13–16, 2009, San Francisco, California, USA.
Copyright 2009 ACM 978-1-60558-371-6/09/04 ...$5.00.

1. INTRODUCTION

As sensor networks mature, we will likely see the emergence of general-purpose sensor network programming systems which provide high-level programming abstractions, support the execution of multiple concurrent applications, and allow applications to dynamically task and re-task the sensor network. These systems achieve their generality by implementing a common suite of services (routing, transport, task dissemination and execution, time synchronization) that can be reused by a variety of applications, thereby greatly simplifying application development. Tenet [9] is an instance of such a system, and has been used for different applications [11, 21].

An important open problem in the context of such systems is energy management. To our knowledge, no existing energy management proposal (and there are many, § 2) has been demonstrated for a programming system of this kind. The key challenge is preserving the generality and wide applicability of such a system while achieving low duty-cycle operation; most existing work in the area makes one or more assumptions (e.g., about the workload or the application's tolerance to latency, about support for broadcast traffic or lack of support for end-to-end reliable transport, and so forth).

In this paper, we discuss the design of a radio duty-cycling approach that we call Application-Informed Energy Management (AEM) (§3). AEM is designed in the context of the Tenet system, and makes two conceptual advances. The first is that Tenet's tasking language permits static analysis of tasks. Static analysis can be used to infer application workload in a manner transparent to the application. The second conceptual advance is to coordinate network-wide radio duty-cycling tailored to the application workload and the expected traffic pattern within the network, using parameters derived from static analysis. This coordination is itself achieved via Tenet's tasking system; AEM dynamically sets up globally synchronized periodic schedules, using the tasking mechanism, for transmission of control (routing, time synchronization, task dissemination) and data packets.

Tenet Requirements	**Supporting Systems**
Low Duty-cycle	SMAC, B-MAC, LPL, SCP-MAC, X-MAC, Koala, Dozer, FPS, AEM
Handle network transients	LPL, AEM
Application Informed	AEM
Multiple applications	FPS, AEM
Time Synchronization	LPL, FPS, AEM
Reliable transport	Koala, AEM

Figure 1—Summary of Tenet requirements and supporting energy management techniques in wireless sensor networks.

It scales the number of data schedules in proportion to the number of concurrent tasks. Moreover, the duration of radio on-times is variable and *elastic*, adapting to load transients and retransmissions. Finally, AEM is designed to be robust to routing changes, time synchronization transients, and node failure.

Using experiments on a 40 node testbed (§5), we show that AEM can achieve duty-cycles of less than 3%, and end-to-end latencies of less than 10s, while at the same time reliably transferring 100% of the generated sensor data, for even very demanding workloads. Moreover, it is highly robust to node failure, adapting gracefully to the failure of half the nodes in a large network. For event-triggered systems that generate minimal data, AEM achieves near 1% duty-cycle, which represents the system overhead required to maintain the generality of the Tenet system. We could have achieved many of our goals using a widely-available duty-cycling MAC protocol such as LPL, but at the expense of high duty-cycles: we show later in the paper that LPL's duty-cycles are 15% or higher for the workloads we examine.

Other energy-management proposals such as Koala [20], Dozer [6], and SCP-MAC [36] achieve lower duty-cycles than AEM. This is not surprising, since they do so at the cost of generality, either assuming a specific workload or specific traffic profile, or lacking support for flexible tasking and concurrent applications. AEM is meant to be complementary to such hand-tuned sleep-scheduling techniques; these should be used where it is necessary to achieve ultra-low duty cycles and it is possible to leverage application knowledge to do so. At the same time, it is important to have a sleep scheduling component like AEM in a general-purpose sensing system to support those application deployments that can live with the (rather substantial) energy savings that AEM provides, but want to enjoy the benefits of a readily available, reusable (and robust) sleep scheduling mechanism.

2. RELATED WORK

Energy management in wireless sensor networks has seen extensive research interest. Much of this research focuses on reducing communication energy by duty-cycling radios, or reducing the volume of information communicated. Since the literature in this area is vast, we focus only on the most closely relevant pieces of work.

Coordinated sleep scheduling uses synchronized sleep-wakeup across all nodes, or subsets thereof. S-MAC [35], T-MAC [31], and SCP-MAC [36], and AppSleep [24] are examples of systems that, after putting a node to sleep, use synchronized wakeup or channel polling often in large groups or the entire network. D-MAC [17], TinyDB [5], Dozer [6], and FPS [12] on the other hand, use staggered sleep schedules such that the parent in a tree routing topology wakes up to receive a packet from its children. Koala [20] coordinates its sleep schedules for bulk transfer. AEM falls into this class of systems, but differs in many respects from all of them: it can support concurrent application tasks, dynamic re-tasking, tailors its sleep scheduling to application needs, and supports system services such as task dissemination, dynamic routing, and time synchronization. Figure 1 summarizes these differences.

Uncoordinated sleep scheduling establishes sleep schedules without any explicit coordination. In transmitter-initiated wakeup schemes such as B-MAC [23], X-MAC [4], and STEM [25], the transmitter sends long preambles or a packet train until the receiver is ready to receive the packet. In the receiver-initiated schemes such as LPP [20] and RI-MAC [28], the transmitter waits for a beacon or signal from the receiver to start data transmission. In Section 5 we perform a quantitative comparison with a transmitter-initiated scheme, LPL [23], and show that AEM can achieve a 6-fold lower duty-cycle.

Application-informed energy management has also been explored in various contexts. In wireless and mobile networks, there are proposals to let the applications configure the power management policies based on their communication requirement [14, 2]. Re-designing OS abstractions that permit applications to achieve energy-efficient I/O is shown to be highly effective in TinyOS [13]. AEM does not explicitly require applications to specify energy requirements, using static analysis instead to infer application workload and to tailor network-wide radio duty-cycling to the workload.

Complementary to AEM is a body of work that relies on a second radio or channel to perform sleep-wakeup coordination. Examples of such *hierarchical power management* systems include PAMAS [27], Paging Channel [1], wake-on-wireless [26], STEM [25], and dual-radio pathway [22]. In contrast, AEM does not require specialized hardware to work.

Tangential to AEM is the literature on two other energy-management topics. First, *redundancy control* attempts to save energy, hence maximize the network lifetime, by turning off nodes that are unnecessary to maintain the desired communication or sensing fidelity. Examples of work in this area include SPAN [7], GAF [34]. Second, *energy aware routing* focuses on the selection of the most energy efficient paths for data delivery [33].

Finally, AEM communication schedules are qualitatively similar to periodic task execution in real-time systems [15] but with one key difference: the end of communication schedule, or communication deadline, is not fixed.

3. DESIGN

AEM is a collection of mechanisms in the Tenet sensing system that enables flexible radio duty-cycling. It analyzes application programs to infer application workload and schedules radio on-times consistent with that workload, thereby achieving low duty-cycle and low latency. In this section, we first describe Tenet, then discuss in detail the design of AEM and its integration with the Tenet architecture.

3.1 Tenet Overview

Motivated by a property common to many recent sensor network deployments [32, 30, 29, 10, 3], we proposed [9] the Tenet architecture for tiered sensor networks. Such networks consist of a lower tier consisting of battery-powered *motes*, which enable flexible deployment of dense instrumentation, and an upper tier containing fewer, relatively less constrained 32-bit nodes with higher-bandwidth radios, which we call *masters*. This paper focuses on mechanisms for achieving energy-efficient operation in the mote tier.

Tenet places application functionality on the resource rich master nodes and provides a generic mote tier networking subsystem that can be reused for a variety of applications, without significant loss of overall system efficiency. This separation of functionality is possible because Tenet disallows multi-node fusion in the mote tier. Instead, any and all communication from a master to a mote takes the form of a *task*, which is a request to perform some activity often based on local sensor values. Any and all communication from a mote is a response to a task, and motes cannot initiate tasks themselves.

Tasks.

A Tenet task is a linear data-flow program consisting of *tasklets*. Motes contain a library of parametrizable *tasklets*, implementing such functionality as timers, sensors, thresholding, data compression, and other forms of simple signal processing. For example, to construct a task that samples the light sensor every 2 minutes and sends the samples to its master, we write:

```
periodic(2 mins)->sample(LIGHT)
  -> Send()
```

The Tenet scheduler maintains a queue of tasks waiting to use the mote's micro-controller. The scheduler operates at the level of tasklets, and knows how to execute the task's tasklets in order. Tenet can execute multiple tasks concurrently.

Services.

The second major component of Tenet is its collection of networking and timing services. Its networking subsystem provides task dissemination from the master to the motes, routing from mote to the masters, and end-to-end reliable transport for applications that require reliable data delivery. Tenet's task dissemination mechanism reliably floods task descriptions in a manner similar to other TinyOS dissemination mechanisms. Its routing subsystem uses any-to-any mesh routing on the master tier. On the mote tier, it computes paths from motes to the nearest master using existing tree-routing implementations (MultihopLQI and CTP). Tenet's reliable transport service uses end-to-end acknowledgments, forwarded along the reverse path from masters to motes, and retransmissions to achieve 100% reliable delivery. Finally, Tenet uses FTSP [18] to provide globally synchronized timing service so that sensor samples can be timestamped for data analysis.

3.2 AEM Goals and Overview

Our design of AEM attempts to achieve the following list of goals:

- Our most important goal is *low duty-cycle operation* – the system must support this in order to ensure network longevity. However, this is not our only goal; we are interested in designs that achieve the lowest possible duty-cycles, while still meeting the other goals listed below. This precludes the use of ultra-low duty-cycle approaches [36, 20], which fail to meet one or more of the goals below.

- Our next goal is *alignment with Tenet's design*. Tenet supports re-tasking, concurrent execution of multiple tasks, flexible forms of tasking (periodic or event-triggered), dynamic routing, end-to-end reliable transport, and time synchronization. We require AEM to support these features as well, since that would extend Tenet's applicability over a wide dynamic range (capable of supporting high data-rate applications like structural monitoring [21] and imaging [11], as well as low-rate applications).

- Our third goal is *robustness*; that AEM should work regardless of changes to the topology, arrival or departure of nodes, or transient failures in other system services such as time synchronization or routing.

- Our fourth goal is *low latency*; AEM should deliver events or samples with a delay no greater than the inter-sample or inter-event time. This requirement is fairly conservative, and in practice AEM does significantly better. However, there is an obvious trade-off between latency and duty-cycles, and we are interested in designs that favor lower duty-cycles over latency.

- Our final requirement is *transparency.* We require that AEM make minimal or no modifications to existing parts of Tenet. This preserves the modularity of the overall system, enabling easy evolution of its components.

AEM achieves these goals using two conceptual advances. The first is based on the observation that Tenet's simple tasking permits static analysis of tasks at the master tier. This static analysis can be used to infer application workload. The second conceptual advance is to tailor radio duty-cycling to the application workload using Tenet's tasking mechanism and some simple functionality built into the mote tier.

We use a simple example to illustrate how this works. In Tenet, when a user wants to collect a light sensor reading every two minutes from each node in the network, the user presents the following task to the system:

```
periodic(2 mins)->sample(LIGHT)
  -> Send()
```

AEM analyzes this task and makes two inferences: (a) that a response will be generated every two minutes and (b) that a response will fit entirely into a single packet with a small payload. These two inferences allow AEM to determine the duration for which radios on the motes should be turned on or off. After performing this analysis, AEM prepends the task with a description of these parameters:

```
dataframe(2mins, ...)->periodic(2 mins)
 ->sample(LIGHT)  -> Send()
```

and disseminates this task into the network. Upon receiving this task, motes schedule packet transmissions and radio activity based on the specified duty-cycle parameters.

Of course, this deceptively simple example hides significant detail, which we describe in the next two sections. In addition to discussing our static task analysis, we also describe how AEM performs robust duty-cycling, while accommodating re-tasking and concurrent task execution.

3.3 Task Static Analysis

A key observation of this paper is that Tenet's design permits inspection of application activity at the master tier. Specifically, in Tenet, we can statically analyze a task just before it is disseminated (and possibly modify it before dissemination) in order to achieve duty-cycling. Since Tenet's tasking language is a linear data-flow language, it is possible to analyze a task and infer the following three parameters, which can be used to control radio duty-cycles: (i) the *start time* when a task starts executing controls when the motes should turn on their radio, (ii) the *period* between two executions of a repeating task determines how often a mote should turn its radio on, and (iii) the *duration* for sending and receiving packets generated in response to the task determines the minimal amount of time its radio should remain on.

To compute these parameters, AEM performs a simple data-flow analysis of the task description, and partitions each task description into the following sections:

```
synchronization -> periodicity
 -> data generation and processing
 -> packing -> send
```

The *start time* is computed by analyzing the synchronization section of the task. Some task descriptions explicitly specify when task execution should start; for example, when data collection needs to be synchronized across all nodes. When this section is missing, AEM takes the liberty to modify the task description to improve system performance. As we describe later, AEM's duty-cycle design ensures that all nodes' wakeup times are synchronized, so synchronizing task execution with these times can result in lower latency.

The *periodicity* section describes how frequently should a mote execute the task and potentially generate data, while the *packing* section describes how many data items should be included in the payload. AEM computes the *period* for duty-cycling using these sections: if a task executes every x seconds and packs n samples into a packet, the period at which packets are generated is $x \times n$.

As we describe below, if one or more of these sections are missing from the task description, AEM makes reasonable choices for the missing section(s). Of course, it is possible to write tasks that do not conform to this template – for example, a task with multiple synchronization and periodicity sections. In this case, our static analysis fails and as a result AEM does not perform duty-cycling. However, we have not come across practical sensing tasks that would be expressed in this fashion.

We now illustrate how AEM's static analysis works for a variety of common task specifications.

Periodic collection: Consider a task that periodically generates data using the light sensor:

```
periodic(1s)->sample(LIGHT)->send()
```

This task is missing the synchronization and packing components. Because the task does not specify an absolute time at which the application should start running, AEM can synchronize the *start times* at all nodes (setting the start time to a future instant, taking dissemination latency into account). If the task description does not specify a packing tasklet, AEM assumes that each sample is sent in a separate packet and derives the *period* from the argument for the `periodic()` tasklet only.

One-shot collection: The following task generates one sample reading:

```
Sample(LIGHT)->send()
```

As before, since this task does not specify a synchronization component, AEM can modify the task to specify a synchronized start time. In the absence of a `periodic()` tasklet, AEM will ensure that the *period* is set to 0.

The following task differs from the above only in that it already specifies a synchronization section:

```
globaltimewait(0x1234abcd)->sample(LIGHT)->send()
```

AEM does not modify the task description in this case, and computes other parameters as described above.

Synchronized periodic collection: The following task requests the motes to sample light every 2 minutes and send 10 samples at a time.

```
globaltimewait(0x1234abcd)->periodic(2 mins)
  ->sample(LIGHT)->pack(10)->send()
```

Following the previous discussion, the *start time* is set to 0x1234abcd, and the *period* is set to 20 minutes because the nodes will send a task response every 20 minutes (10 samples are packed into one packet).

Event-triggered collection: Tenet allows users to specify event-triggered data collection. For example, the following task generates a packet if the light readings exceed a threshold of 10:

```
periodic(1000ms)->sample(LIGHT)
  ->threshold(LIGHT, 10)->send()
```

AEM can not know in advance if the readings will exceed the threshold so it conservatively assumes that each sample will exceed the threshold. All the parameters are thus derived in the same way it is derived for a periodic collection. However, as we describe later, AEM incurs only a small overhead for this conservative approach: if, when the radio is turned on, AEM observes no activity, it quickly turns the radio off. In this way, even for event-triggered collection, near 1% duty-cycles are possible.

In general, the freedom to analyze and modify task descriptions presents several optimization opportunities. For example, AEM can schedule task execution more precisely, so that data is generated *just before* the radio turns on, enabling more efficient use of radio on time. Similarly, AEM can schedule start times of multiple tasks so their radio on-times can overlap, reducing the energy expended in turning the radios on or off. Finally, it is possible to control network parameters such as the transport-layer timeout values using the results of static analysis. We have left these to future work.

3.4 Dutycycling

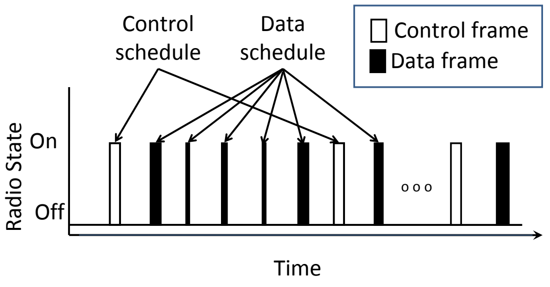

Figure 2—Radio Dutycycling Frames and Schedules.

The second contribution of our work is a mechanism for network-wide radio duty-cycling based on the parameters computed from task analysis, as described above. Although we have described task analysis for a single task, the duty-cycling mechanism should allow for concurrent tasks, and accommodate traffic generated by system services such as routing, dissemination, and time synchronization.

In AEM, the master nodes use the results of task analysis to compute and distribute duty-cycling information to the motes. AEM's duty-cycling mechanism falls into the class of schemes that perform *scheduled* wakeup — radios are turned on and/or off at pre-determined times. This is a natural choice, since we are able to infer application workload through static analysis.

Before discussing the design of AEM's scheduled wakeup, we introduce two terms. A *frame* is a time interval during which the radio at a specific node is on (i.e., capable of transmission or reception). A *schedule* consists of a (usually periodic) sequence of frames.

AEM uses *synchronized elastic frames*. The start time of each frame is pre-determined, but its end is not. Instead, a frame ends (and the radio is turned off) when no packets are detected in the channel for a specified period of time. This approach is robust to topology changes, does not require pre-computation of frame durations, and can absorb transient traffic fluctuations while still achieving low duty-cycles.

Frames and Schedules in AEM .

An AEM master node computes schedules before disseminating them into the network. Each schedule has three parameters t, l, p. The radio at every node is first turned on at time t, and then again every p seconds. Thus, at each node, the radio is turned on at $t + p$, $t + 2p$, and so forth; these times mark the beginning of successive frames. While the radio is on, nodes can receive and transmit packets. In particular, each node contends with other nodes (using a CSMA MAC) to transmit packets. We require no modifications to the MAC layer.

An important feature in the design of AEM is the parameter l, called the *quiet-time*[1] It determines the amount of time for which the radio stays on after the last received or transmitted packet. At the end of this time, the radio is turned off. Thus, in AEM, frame lengths are not fixed, but are elastic, with a minimum length of l. Frames adapt to the activity in the channel, permitting the system to handle load transients or increases in the number of retransmissions. Load transients might occur, for example, when a routing change causes packets to be backed up at a node; after these routing changes are resolved, the packets can be transmitted during a frame.

Since AEM is required to conform to the Tenet design, it needs to support the transmission of control traffic in addition to data traffic resulting from possibly multiple concurrent tasks. To support control traffic and multiple concurrent tasks, AEM: a) distinguishes between two types of schedules, control and data schedules (Figure 2); b) allows multiple schedules to be active at any given time.

Control traffic is sent during the control schedule and data traffic is transmitted during the data schedule. This way, control traffic is isolated from data traffic since the loss of control traffic can adversely affect system performance. The data schedule parameters are derived from task analysis. t is derived from the *start time* parameter. The parameter p can simply be computed from the *periodicity* parameter derived from task analysis. However, this has latency implications, especially for reliable transport protocols that use end-to-end acknowledgments. For this reason, to enable fast end-to-end retransmissions of data packets, AEM trades-off some duty-cycle for reduced latency by scheduling data frames more frequently than the *periodicity* value would suggest.

For simplicity, the control schedule is a sequence of periodic control frames with fixed periodicity, which is usually statically determined based on control protocol parameters. Many control protocols (e.g., routing and time synchronization) are naturally periodic, but some others (e.g., Tenet's task dissemination, which uses Trickle [16] style exponential timers) are not.

Once schedules are computed, they are disseminated to nodes using Tenet's task dissemination mechanism. There exist specific tasklets that can be used to specify control and data schedules. A Tenet master node can generate and disseminate a task description that contains multiple tasklets specifying the current set of schedules. These can be disseminated even when the network is already being duty-cycled using a different set of schedules; thus, in AEM, we can re-schedule duty-cycling (to, for example, accommodate a new task injected into the system). Using the task dissemination mechanism for controlling duty-cycling has another advantage: when a task is deleted from the system, its corresponding data schedule can also be deleted using the same mechanism.

AEM schedules are a natural fit for periodic traffic patterns. AEM schedules also support tasks that send event-triggered responses (e.g., when a sensor reading exceeds a certain threshold), with a slight loss of efficiency. Nodes turn their radios on in synchrony, but when there are no events, they are turned off after *quiet-time*. As we show in our experiments, we are able to achieve low duty-cycles even with this slight loss of efficiency. However, AEM's periodic control schedules are not a perfect match for some control protocols which use exponential timers to schedule control packet transmissions. One example is Tenet's task dissemination mechanism. It works well in AEM, at the cost of slightly higher task dissemination times. However, CTP [8] control traffic does not work well in AEM; CTP uses aggressive beaconing to quickly detect and repair loops. We have left the integration of CTP with AEM to future work, but we note that CTP was not explicitly designed to support schedule wakeup schemes.

Finally, as we have discussed before, multiple concurrent schedules can be active in the system at any given time. More precisely, at any time, there will be one active control schedule, and zero or more data schedules. It is therefore possible that two frames belonging to two different schedules may overlap; while one frame is active, the start time for a second frame may occur. AEM handles this easily by keeping the radio on, and enabling transmissions for the second frame. These transmissions contend with any remaining transmissions from the first frame, and the frame duration is extended until no activity is detected.

Performance Implications.

Our design has two interesting performance implications. First, in this design, node transmissions are synchronized to the beginning of a frame. This increases the likelihood of packet loss relative to duty-cycling designs that do not use scheduled wakeup. Data transmissions are resilient to these packet losses because of link layer retransmissions. However, broadcast control packets are affected more significantly because of this synchronization. Losses in control packets can delay routing convergence, or cause nodes to be de-synchronized. In AEM, we alleviate the loss of control packets by spreading control transmissions across different successive con-

[1]AEM's *quiet-time* is similar to LPL's off-timer, but has a different function. LPL's off-timer is designed to allow transmission of back-to-back packets without duty-cycling the radio in between. In AEM, the *quiet-time* is used to extend a frame to handle load transients.

trol frames. For example, if the routing protocol sends one beacon every 30s, we set up control frames that repeat every 15s and allow half the nodes (e.g., those with even node IDs) to transmit during the first frame and the other half during the second frame. All nodes have their radios on during both frames, so they can receive all transmissions. This technique improves control traffic reliability at the expense of additional radio on-time. More generally, the number of additional frames should adapt to network density, and we have left this adaptation to future work.

The second performance implication is more subtle, and arises from our design of elastic frames. Consider a chain topology A-B-C. Node C might not overhear the packets transmitted by A to B and as a result it might put its radio to sleep after *quiet-time*. When B starts forwarding the packets to C, its packets are dropped, reducing overall efficiency and delivery ratio. Fortunately, this scenario happens only occasionally, and there is a simple solution. In our example, B has to determine when it should stop transmitting to C. B infers that C's radio might already be off if these two conditions are true:

- The *quiet-time* has elapsed without B having received a packet or an acknowledgment from C.
- B did not receive an acknowledgment from C even after a fixed number (5 in our implementation) of consecutive retransmissions.

When these conditions are met, B pauses forwarding packets to C. The second condition is conservative, since C's radio might be on but the link from B to C might be lossy. Even in this case, it is better to pause forwarding until the next data frame since that might allow us to later resume forwarding on a better link when the routing protocol recomputes the routes.

Bootstrapping.

When the network starts, all radios are turned on. A master node, perhaps under the control of the system administrator, can disseminate a control schedule to begin radio duty-cycling. The users utilize the Tenet tasking mechanism to initiate duty-cycling. Once this task is disseminated, duty-cycling operation can start. While a network is in duty-cycled mode, tasks can be disseminated, as can additional data schedules.

When a node joins the network, it keeps its radio on. However, at this point it does not know anything about the schedules active in the network. It will eventually learn this from Tenet's task dissemination mechanism. When it does, and after it is time synchronized with the rest of the network, the node can start duty-cycling its radios. Until then, it behaves just as it would if it were not duty-cycled, with one important exception: it queues all control packet transmissions, until it overhears another control packet transmission (and likewise for data packets), at which point it attempts to clear the corresponding queue. This *frame inference* technique ensures that it participates correctly in the duty-cycling schedules, even though it has no explicit knowledge of the schedules. This conservative approach works even when a group of topologically contiguous nodes joins the network, as long as some subset of the nodes are operating consistent schedules or their transmissions are triggered by transmissions from a master node.

Handling Time Synchronization Failures.

AEM relies on a network time synchronization protocol, like FTSP [18]. Such protocols suffer from clock drifts and AEM allows for this by using a small guard time (2ms, twice the largest synchronization error we have seen in our network) before transmitting data packets at the beginning of each frame.

However, network time synchronization can fail in more pathological ways. If a few FTSP beacons are lost, a node can become de-synchronized from the rest of the network. We have seen this occur rarely in our experiments, but it is important to design AEM to be robust to such de-synchronizations. When a node is desynchronized (i.e., FTSP signals a loss of synchronization), AEM puts the node in "recovery" mode. In this mode, it turns on the radio and keeps it on until the node is synchronized again. Because the radio is on, the node can receive all the packets sent by the neighbors, including time synchronization beacons. This allows the time synchronization protocol to improve its estimate and stabilize. However, during recovery, a node might still be in the forwarding path. It can still receive packets, but cannot transmit unless it knows that other nodes' radios are on. To send packets, it uses the frame inference technique described above: it queues all control packet transmissions, until it overhears another control packet transmission (and likewise for data packets), at which point it attempts to clear the corresponding queue. This technique generalizes easily to the case when many nodes are de-synchronized: eventually, all nodes will be "clocked" by transmissions from the master node.

Alternative Schedule Designs.

There are many possible designs for a scheduled wakeup scheme that satisfy our goals stated earlier, and we designed and implemented two alternatives before settling on the design we present in this paper.

Staggered Frames: In this design, the transmission and reception schedules are staggered along the routing tree in such a way that a parent's frame overlaps with that of its children, and the parent waits to receive packets from all its children before transmitting to its own parent. This approach (similar to DMAC [17]) is efficient but is fragile. It adapts poorly to network topology changes – sometimes a single parent change can result in re-computation and resynchronization of schedules in half of nodes in the network. Moreover, it requires additional control overhead to adapt to such changes.

Fixed Frames: In this design, frame start and end times are synchronized across the network. Because the frame setup does not explicitly use any topology information, this scheme is robust to routing topology changes. However, this scheme requires an accurate precomputation of frame duration (the previous scheme suffers from this drawback as well).

4. IMPLEMENTATION

We have implemented AEM as a component of Tenet in TinyOS 2.x. It uses in 2.7 KB of code space and requires 128 bytes of RAM for task and duty-cycling state maintenance. In addition, AEM also requires RAM for packet buffers, as we discuss below. We use buffer size of 6 packets in our experiments. AEM's schedules can be configured and altered using the Tenet tasking mechanism, as discussed in §3.

While designing AEM, one of our goals was to leverage as much of the existing Tenet software as possible (§3.2). Our implementation is able to achieve this, with two exceptions noted below. AEM transparently interposes an asynchronous packet buffer to which MultihopLQI routing protocol, FTSP time synchronization protocol, and the Tenet dissemination protocol send their packets. The AEM module orchestrates the packet egress from this buffer depending on the duty-cycling state of the radio. Data packets use a similar but separate queue. Upon a successful packet enqueue operation, the Tenet stack progresses as if the packet transmission operation had been completed in a non-duty-cycled network.

We made three modifications to existing software components. First, we increased the end-to-end retransmission timeout for the Tenet packet transport protocol for AEM. Such a change is required for any duty-cycling method which increases packet latency. Second, AEM required one change in the MultihopLQI forwarding engine to pause transmissions when it guesses that the receiver's radio might be turned off (§3.4). Finally, we added frame inference (§3.4) to the base station to time its transmissions. The alternative would have been to run a complete instance of AEM at the base station. Our implementation preserves the transparent bridging design of the base station.

5. EVALUATION

In this section we evaluate, through experiments conducted on a 40-node testbed, AEM's ability to achieve low radio duty-cycle and data delivery latency while preserving all the design requirements of Tenet.

5.1 Methodology

We conducted our experiments on a tiered network testbed with several Stargate nodes and 40 TelosB motes. All nodes are located above the false ceiling across multiple rooms and hallways on a floor (50m by 20m area) of a large office building. The wireless environment above the false ceiling is harsh, with some links experiencing above 30% packet loss rates. All nodes run the Tenet stack modified to support AEM. In most experiments, we use a single Tenet master node. We configured the mote radios to transmit at -8.906 dBm, which results in a tree with 4-hop depth.

In our experiments, we are interested in measuring the steady state behavior of AEM. For this reason, each experiment starts with a 10 minute initialization period during which routing paths are established, time is synchronized, dissemination states are initialized, and control schedules are set up[2]. Measurements start 10 minutes into the experiment, when an application injects a task and the network initiates data schedules.

Our experimental workload is as follows. In most experiments, a fixed fraction f of the nodes are tasked to generate a sensor reading once every 2 minutes. When f is small, our setup approximates an event-triggered workload, and when f is large, a periodic workload. Unless otherwise stated, each run of an experiment lasts for 40 minutes, and all experiments are averaged over 3 or more runs.

In each experiment, the nodes are tasked to use Tenet's end-to-end reliable transport mechanism. Thus, in every experiment, *the delivery ratio is 100%.*

During each experiment, we measure:

Duty-cycle: We compute duty-cycle by dividing the total time the radio is on at each node by the experiment duration. We are interested both in the average duty-cycle across all nodes, and (in some cases) the distribution of duty-cycles.

Latency: We measure the elapsed time between packet generation at a mote and packet arrival at the master. As with duty-cycle, we are interested both in average and distributional performance.

These metrics correspond to two of the goals described in §3.2. We also have designed experiments to measure AEM's adherence to other goals. To demonstrate AEM's alignment with Tenet's design, we conduct experiment with multiple concurrent tasks, and another with multiple master nodes. To demonstrate its robust-

[2]Later in this section, we present one experiment that demonstrates that our AEM implementation correctly adapts to large transients.

ness, we conduct an experiment where half the nodes in the network are made to fail, showing that AEM recovers. Finally, the transparency goal is achieved by careful implementation (§4).

5.2 Comparisons

Aside from other approaches to duty-cycling listed in §3.4, there is one other plausible duty-cycling approach that would have satisfied many of our goals. LPL[3] is the only radio duty-cycling software system available that is mature and robust enough to support Tenet requirements. We integrated LPL into the Tenet stack. In this section, we also compare AEM against Tenet with LPL. The goal of this comparison is to test whether an existing design would have sufficed.

To calibrate AEM's performance, we compare its duty-cycle with that of an *omniscient scheduler*. An omniscient scheduler only keeps the radio on for the exact amount of time necessary to send or receive all the packets transmitted during an experiment (this includes all control and data packets as well as retransmissions). We compute the omniscient scheduler's duty cycle by counting all the transmissions and receptions at each node during an experiment, assigning a nominal average transmission and reception time (10ms, derived experimentally), and dividing the total time by the experiment duration. AEM itself deviates from this scheduler because its guard time and its quiet time constitutes overhead.

5.3 Single Task Performance

We first explore AEM's performance when the user executes one task in the network. We vary the fraction f of nodes that execute the task, presenting a varying workload to the system. For each fraction f, nodes are selected uniformly from across the network.

To study the impact of workload on duty-cycle and latency, we do experiments with six different workloads with 0% to 100% of the nodes responding to the injected task. Figure 4 illustrates the control and data schedules during one of our experiments at three nodes along a path in the routing tree. AEM achieves low duty-cycle by turning on the radio only during these schedules. Notice how frames are aligned, and how some frames are longer than others, demonstrating elasticity. Finally, notice how the length of elastic frames increases as we go up the tree: the duty-cycle adapts to increasing traffic automatically.

Figure 5 shows that AEM achieves a duty-cycle of about 1.6% when there is no task response and about 2.7% when all 40 nodes respond to a task with sensor data. This performance is remarkable: when all 40 nodes respond, the network is generating one packet on average every 3 seconds, yet the system is able to maintain a 2.7% duty-cycle. This performance is within a factor of 2-3 of the omniscient scheduler. This difference is not surprising, since most of AEM's inefficiency comes from its 70ms *quiet-time* (the duration of 7 packet transmissions). This choice is rather conservative, and we expect to be able to optimize it significantly.

Figure 8 shows the distribution of duty-cycle across the nodes with AEM when 8 out of 40 nodes respond to the task. (The results are similar for other fractions of responding nodes). The distribution is tight, with a range from 1.6% to 4.9% and a 90th percentile duty-cycle at 3.7%.

Finally, Figure 6 shows that the data delivery latency ranges from 5.8s to 13.9s. AEM, with its elastic frames, tries hard to transmit a packet from sender to base station within one data frame duration. However, if a packet needs to be retransmitted, the sender needs to wait for a transport timeout (15s in our implementation) and the next data frame (the data frame periodicity is 10s). Retransmitted packets explain why the latency is higher than the forwarding latency, and also why the latency varies significantly between experiments. Figure 7 shows that the latency distribution ranges from 0.4s to 16.5s across the nodes with 90th percentile latency at 11s.

LPL comparison.

Figure 5 shows that LPL duty-cycles are a factor of 6-9 higher than AEM and a factor of 18-19 higher than the omniscient scheduler. This is attributable to LPL's high system maintenance overhead (15.2% vs 1.6% for AEM with no data packets to send) dominated by broadcast traffic which is especially costly for LPL due to long preambles. This also explains why LPL's duty-cycle is relatively insensitive to workload. Figure 3 illustrates how this problem can result in the radio being turned on for up to half a second at a time in our experiments. Finally, Figure 8 shows that, with 8 out of 40 nodes responding to a task, the duty-cycle for LPL can range from 11% to 21%, a much larger range than with AEM. Because of its higher duty-cycle, LPL is able to achieve much lower latencies than AEM; Figure 7 shows the latency distribution to be in the 0-5s range.

Thus, AEM meets our duty-cycle and latency design goals. LPL satisfies most of the other goals, with the exception of low-duty cycle operation (our primary goal). In the following experiments, we illustrate AEM's adherence to our other goals.

5.4 Multiple Task Performance

AEM reacts to a new task insertion (potentially by different users) by setting up an additional data sched-

[3]We use BoX-MAC-2 [19]included in current TinyOS 2.x version as a stable implementation of LPL.

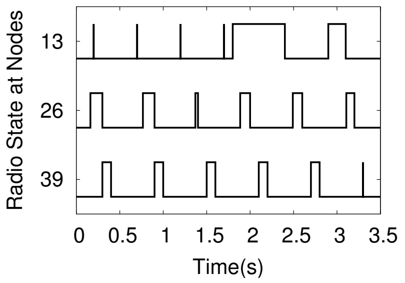

Figure 3—Radio states across nodes with LPL during an experiment.

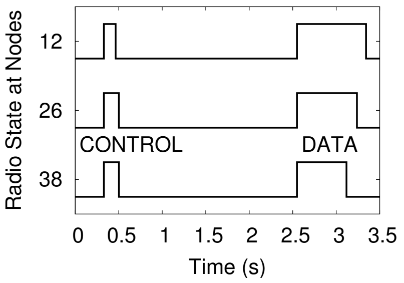

Figure 4—AEM control and data frames during an experiment.

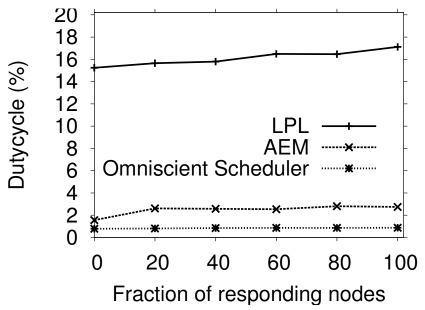

Figure 5—AEM and LPL duty-cycle with varying workloads.

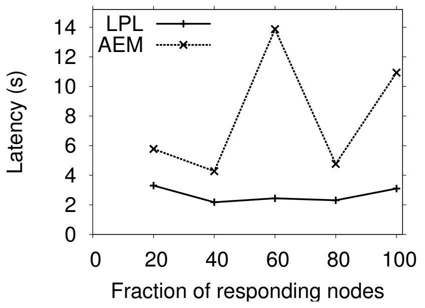

Figure 6—Task Response Latency with different workloads.

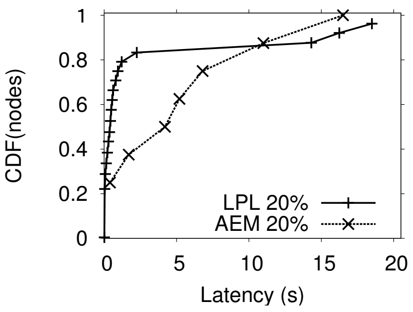

Figure 7—Latency distribution with AEM and LPL.

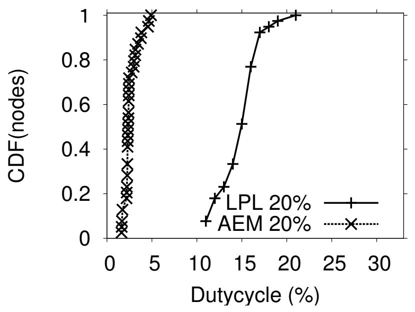

Figure 8—Duty-cycle distribution with AEM and LPL.

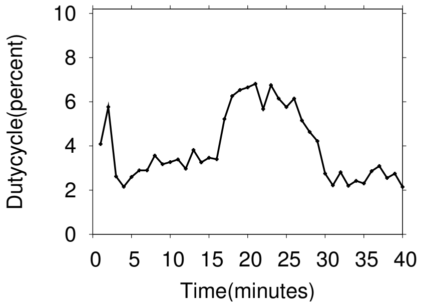

Figure 9—Dutycycle increases with additional tasks and decreases as tasks terminate.

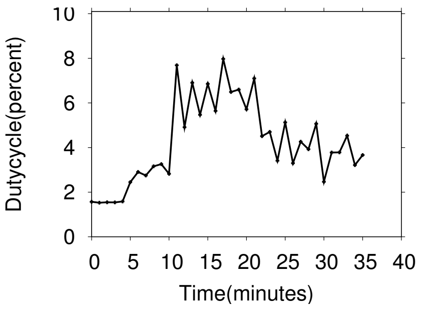

Figure 10—50% of the nodes fail at 20 mins.

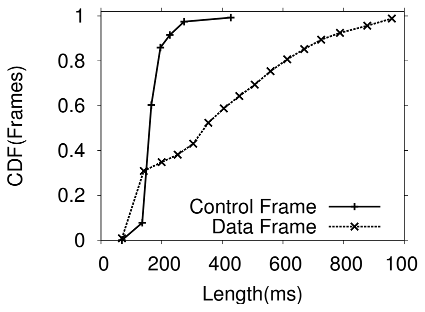

Figure 11—Distribution of Control and Data frame sizes.

ule to accommodate the traffic generated in response to the task. To study how AEM adapts to task insertion and deletion, we conducted the following experiment: the first task is injected 5 mins after the system initialization, the second task at 15 mins, the second task is terminated at 25 mins and the first task is terminated at 35 mins. The first task is a periodic task that generates sensor readings every 2 mins but filters data locally so that only 20% of the nodes respond with data. The second task is also a similar periodic task with the same data filter but it generates sensor readings every minute.

Figure 9 shows the 1-min windowed average dutycycle across the nodes over time. The dutycycle increases with the insertion of the first task, increases further with the insertion of the second task, decreases when the second task is terminated and decreases further when the first task is deleted. Thus AEM performance adapts to multi-task scenario as expected: higher duty-cycles with increased number of tasks.

5.5 Tiered Networks

Tenet is designed for tiered networks, so AEM must support such networks as well. Our design for AEM required no changes in order to support multiple masters. We conducted a single-task experiment using the same 40 nodes, but with two masters. As expected, the duty-cycle decreases (from 2.7% with one master to 2.4% with two) since the traffic is now spread out over two trees. However, the decrease is not dramatic, since nodes in one tree can still overhear some nodes in the other tree. We see no noticeable change in latency; this is because the dominant component of latency in AEM is the transport timeout and the data periodicity, not the forwarding latency.

5.6 Robustness

AEM is, by design, robust to transient loss of desynchronization of network time, and to routing dynamics. It is also robust to packet loss (since its schedule dissemination re-uses Tenet's task dissemination mechanism). In this section, we conduct an experiment to demonstrate AEM's robustness to node failure. During the course of a 40-node single-task experiment, we deactivated 50% of the network nodes. In this experiment, we used full transmit power, so that the deployment was dense enough that the rest of the network remained fully connected. Figure 10 plots the per-min average duty-cycles across the network as a function of time. As expected, the duty-cycle increases when the task is first injected into the system at 10 mins. Between 25 and 30 mins into the experiment, we deactivated 20 nodes. As the figure shows, AEM continues to work despite this disruption, and has an average duty-cycle that is about half of what it was before the node failures, as one might expect.

5.7 Other experiments

Finally, we briefly discuss several other experiments that give us insights about AEM's performance or explore the sensitivity of our results to parameter settings.

Frame length distribution.

Figure 11 shows the distribution of the length of the frames during an experiment in which all the motes responded to a single task. The minimum possible frame length is 70ms (the value of the *quiet-time* parameter). It is interesting to note that many data frames are significantly longer than the minimum frame length, illustrating that the system adapts when necessary to absorb retransmissions. Control frames are generally smaller, since control packets are not retransmitted and their load does not vary with time.

Performance at a higher density.

In all our experiments, we have conducted experiments with the same radio transmit power setting. To validate that AEM's duty-cycle is still low at higher densities, we conducted an experiment where all the motes used 0 dBm transmit power, and all were responding to a single task. At this higher density, we observed that the average duty-cycle decreased from 2.7% to 2.26%, and the latency from 10.6ms to about 9.2ms. At the higher transmit power, the shallower tree results in lower duty-cycles (since fewer nodes forward the packet), and better quality paths cause fewer retransmissions resulting in slightly lower latency.

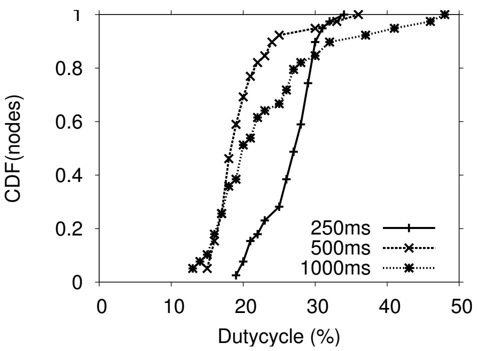

Figure 12—LPL experiment results for varying sleep interval in our 40-node testbed running Tenet stack

LPL's Parameter Sensitivity.

LPL performance is sensitive to its sleep interval and channel polling count threshold. In our experiments, we used a 500ms sleep interval. Figure 12 shows that, for the workloads we use, 500ms sleep interval gives low duty-cycles. With a 1000ms sleep interval, the preamble length overhead outweighs the polling overhead, while with a 250ms sleep interval, the opposite is the case.

LPL performs a sequence of Clear Channel Assessments to check if the channel is clear. If it detects more CCA samples than a specified threshold, it assumes the presence of a transmitter and keeps the radio on to receive a packet. In our experiments, we used the default value of 3. A small value can cause many false positives in channel assessment, causing the node to turn on a radio more frequently. On the other hand, a larger value might miss packets, resulting in higher loss rates. With a threshold of 30, LPL's average duty-cycle is reduced from about 19% to about 16%. More experiments might be needed to find the "optimal" threshold for a given environment, but we believe that our general conclusion (that AEM provides lower duty-cycle operation) holds.

6. CONCLUSION

General purpose sensor network programming systems that provide high level of programming abstraction and allow dynamic execution of multiple and concurrent application have the potential to enable rapid deployment of sensor network applications. Often the radio duty-cycling protocols that are designed and optimized for specific workloads are not extensible to such systems. In this paper, we have presented AEM, an energy management system that duty-cycles the sensor network based on a static analysis of Tenet applications. The duty-cycle scheduling is also designed to accommodate transient and unexpected changes in traffic due to network dynamics. Our experience with AEM suggests that it is possible to run a general purpose, interactive, and dynamically taskable multi-user networks such as Tenet at sub 3% duty-cycles for widely varying workloads.

7. REFERENCES

[1] Y. Agarwal, C. Schurgers, and R. Gupta. Dynamic power management using on demand paging for networked embedded systems. In *ASP-DAC '05*, pages 755–759, New York, NY, USA, 2005. ACM Press.

[2] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning Wireless Network Power Management. In *MobiCom '03*, pages 176–189, New York, NY, USA, 2003. ACM Press.

[3] A. Arora. ExScal: Elements of an Extreme Scale Wireless Sensor Network. In *ERTCSA '05*, August 2005.

[4] M. Buettner, G. V. Yee, E. Anderson, and R. Han. X-MAC: A Short Preamble MAC Protocol for Duty-Cycled Wireless Sensor Networks. In *SenSys '06*, page to appear, Boulder, Colorado, USA, November 2006. ACM.

[5] P. Buonadonna, J. Hellerstein, W. Hong, D. Gay, and S. Madden. TASK: Sensor Network in a Box. In *EWSN '05*, Istanbul, Turkey, January 2005.

[6] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: ultra-low power data gathering in sensor networks. In *IPSN 2007*, pages 450–459. ACM, 2007.

[7] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris. Span: An Energy-Efficient Coordination Algorithm for Topology Maintenance in Ad Hoc Wireless Networks. In *ACM Mobicom '01*, pages 85–96, Rome, Italy, July 2001.

[8] O. Gnawali, R. Fonseca, K. Jamieson, and P. Levis. CTP: Robust and Efficient Collection through Control and Data Plane Integration. *Stanford Information Networks Group Technical Report SING-08-02*, 2008.

[9] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET Architecture for Tiered Sensor Networks. In *ACM SenSys '06*, Boulder, Colorado, USA, November 2006.

[10] R. Guy, B. Greenstein, J. Hicks, R. Kapur, N. Ramanathan, T. Schoellhammer, T. Stathopoulos, K. Weeks, K. Chang, L. Girod, and D. Estrin. Experiences with the Extensible Sensing System ESS. *CENS Technical Report 61,* March 29 2006.

[11] J. Hicks, J. Paek, S. Coe, R. Govindan, and D. Estrin. An Easily Deployable Wireless Imaging System. In *ImageSense 2008*, 2008.

[12] B. Hohlt and E. Brewer. Network Power Scheduling for TinyOS Applications. In *IEEE ICDCSS '06*, San Francisco, California, USA, June 2006.

[13] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP 2007*, pages 251–264, New York, NY, USA, 2007. ACM.

[14] R. Kravets and P. Krishnan. Application-driven power management for mobile communication. *Wireless Networks*, 6(4):263–277, 2000.

[15] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrarydeadlines. In *IEEE RTSS 1990*, Lake Buena Vista, FL, USA, 1990. IEEE Press.

[16] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *USENIX NSDI '04*, San Francisco, California, USA, 2004.

[17] G. Lu, B. Krishnamachari, and C. S. Raghavendra. An adaptive energy-efficient and low-latency MAC for data gathering in sensor networks. In *WMAN '04*, 2004.

[18] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *ACM SenSys '04*, pages 39–49, New York, NY, USA, 2004. ACM Press.

[19] D. Moss and P. Levis. BoX-MACs: Exploiting Physical and Link Layer Boundaries in Low-Power Networking. *Stanford Information Networks Group Technical Report SING-08-00*, 2008.

[20] R. Musaloiu-E., C.-J. M. Liang, and A. Terzis. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *IPSN 208*, pages 421–432, Washington, DC, USA, 2008. IEEE Computer Society.

[21] J. Paek, O. G. K.-Y. Jang, D. Nishimura, R. Govindan, J. Caffrey, M. Wahbeh, and S. Masri. A Programmable Wireless Sensing System for Structural Monitoring. In *4th World Conference on Structural Control and Monitoring(4WCSCM)*, San Diego, CA, July 2006.

[22] T. Pering, V. Raghunathan, and R. Want. Exploiting Radio Hierarchies for Power-Efficient Wireless Device Discovery and Connection Setup. In *VLSID '05*, pages 774–779, Washington, DC, USA, 2005. IEEE Computer Society.

[23] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *SenSys 2004*, pages 95–107, Baltimore, Maryland, USA, 2004. ACM Press.

[24] N. Ramanathan, M. Yarvis, J. Chhabra, N. Kushalnagar, L. Krishnamurthy, and D. Estrin. A Stream-Oriented Power Management Protocol for Low Duty Cycle Sensor Network Applications. In *EmNetS-II '05*, Sydney, Australia, May 2005.

[25] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. Srivastava. Topology management for sensor networks: exploiting latency and density. In *MobiHoc '02*, pages 135–145, New York, NY, USA, 2002. ACM Press.

[26] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless:: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *MobiCom '02*, pages 160–171, New York, NY, USA, 2002. ACM Press.

[27] S. Singh and C. S. Raghavendra. PAMAS: power aware multi-access protocol with signalling for ad hoc networks. *SIGCOMM Comput. Commun. Rev.*, 28(3):5–26, 1998.

[28] Y. Sun, O. Gurewitz, and D. B. Johnson. RI-MAC: A Receiver Initiated Asynchronous Duty Cycle MAC Protocol for Dynamic Traffic Load. In *ACM SenSys '08*, Raleigh, North Carolina, USA, November 2008.

[29] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons From A Sensor Network Expedition. In *EWSN '04*, Istanbul, Turkey, Jan. 2004.

[30] G. Tolle, J. Polastre, R. Szewczyk, N. Turner, K. Tu, P. Buonadonna, S. Burgess, D. Gay, W. Hong, T. Dawson, and D. Culler. A Macroscope in the Redwoods. In *ACM SenSys '05*, San Diego, California, USA, November 2005.

[31] T. van Dam and K. Langendoen. An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks. In *Proceedings of the ACM SenSys Conference*, pages 218–229, Los Angeles, California, USA, November 2003. ACM.

[32] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *USENIX OSDI '06),* Seattle, Washington, USA, November 2006.

[33] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *ACM SenSys '03*, pages 14–27. ACM Press, 2003.

[34] Y. Xu, J. Heidemann, and D. Estrin. Geography-informed energy conservation for Ad Hoc routing. In *MobiCom '01,* pages 70–84, New York City, New York, USA, 2001. ACM Press.

[35] W. Ye, J. Heidemann, and D. Estrin. Medium Access Control with Coordinated, Adaptive Sleeping for Wireless Sensor Networks. *ACM/IEEE Transactions on Networking,* 12(3):493–506, June 2004.

[36] W. Ye, F. Silva, and J. Heidemann. Ultra-Low Duty Cycle MAC with Scheduled Channel Polling. In *Proceedings of the Fourth ACM SenSys Conference*, Boulder, Colorado, USA, November 2006. ACM.