Network Power Scheduling for TinyOS Applications

Barbara Hohlt and Eric Brewer

Electrical Engineering and Computer Sciences Department
University of California at Berkeley
Berkeley, CA USA
{hohltb, brewer}@eecs.berkeley.edu

Abstract. This paper presents a study of the Flexible Power Scheduling protocol and evaluates its use for real-world sensor network applications and their platforms. FPS uses dynamically created schedules to reserve network flows in sensor networks allowing nodes to turn off their radio during idle times. We show that network power scheduling has high end-to-end packet reception and can achieve power savings of 2-5x for two well-known TinyOS applications over their existing power-management schemes, and over 150x compared with no power management. Twinkle is our second-generation implementation of FPS and provides additional application support.

1 Introduction

Power is one of the dominant problems in wireless sensor networks. Constraints imposed by the limited energy stores on individual nodes require planned use of resources, particularly the radio. Sensor network energy use tends to be particularly acute as deployments are left unattended for long periods of time, perhaps months or years. Communication is the most costly task in terms of energy [2,9,27,21]. At the communication distances typical in sensor networks, listening for information on the radio channel is of a cost similar to transmission of data [23]. Worse, the energy cost for a node in idle mode is approximately the same as in receive mode. Therefore, protocols that assume receive and idle power are of little consequence are not suitable for sensor networks. Idle listening, the time spent listening while waiting to receive packets, comprises the most significant cost of radio communication. Even for hand-held devices Stemm et al. observed that idle listening dominated the energy costs [30]. Thus, the biggest single action to save power is to turn the radio off during idle times.

Unfortunately, turning the radio off implies that you must know that the radio will be idle in advance, and the easiest way to do this is to have a schedule. An obvious approach is to use TDMA to turn the radio off at the MAC layer during idle slots. However, this requires tight time synchronization and typically hardware support. Scheduling network *flows* helps for multi-hop topologies, which play a significant role in wireless sensor networks. Pottie and Kaiser [21] cover the many advantages of multi-hop topologies, including reduced energy use and routing around obstructions. In multi-hop networks the farthest nodes have more chances to drop packets, and thus using only hop-by-hop decisions (rather than flows), as with any MAC-layer approach, tend to achieve lower bandwidth and less fairness.

Flexible Power Scheduling (FPS) [13] introduced the approach of scheduling the network for power savings in sensor networks and proposed a two-level architecture that combines coarse-grain dynamic scheduling at the network layer to plan radio on-off times, and simple CSMA to handle channel access at the MAC-layer. The FPS paper presented the distributed scheduling algorithm details and microbenchmarks, but no performance evaluation with real applications.

In this paper we present a study of the FPS protocol and evaluate its use for real-world sensor network applications with studies of two well-known sensornetwork applications, GDI and TinyDB, on three mote platforms, `mica`, `mica2dot`, and `mica2`. Our second-generation implementation of FPS, named Twinkle[1], is used in these studies. We compare the power savings of GDI and TinyDB running their default radio power management against these two applications running Twinkle radio power management.

The main contribution of this paper is the implementation and evaluations from two real applications using Twinkle, our second-generation implementation of FPS. In particular, we provide an application-level evaluation of the power savings using two well-known and deployed TinyOS applications [11]: the Great Duck Island [18,31] deployment and a TinyDB application that collects data on Redwood trees [17, 29]. We also compare Twinkle with low-power listening, an alternative proposal for power savings.

The contributions of this paper include:

- An implementation and evaluation of network power scheduling with two well-known TinyOS applications accross three platforms
- A 4x power savings for the Great Duck Island application.
- A 4.3x power savings for a 35-mote sensor network using TinyDB, compared with the default "duty cycling" power management scheme, and 150X versus no power management.
- A detailed comparison between Twinkle and Low-Power Listening with measured power data from real motes. This reveals a 2x or more power savings due to Twinkle.

Section 2 presents an overview of the basic FPS scheduling approach to provide background for these studies. Section 3 and Section 4 present evaluations using two real applications. Finally, Section 5 covers related work, and we conclude in Section 6.

2 Background

Flexible Power Scheduling (FPS) [13] introduced the approach of scheduling the network for power savings in sensor networks and proposed a two-level architecture that combines coarse-grain dynamic scheduling at the network layer to plan radio on-off times, and simple CSMA to handle channel access at the MAC-layer. The original protocol only supported communication in one direction, from the network to the gateway. Twinkle is our second-generation implementation of the FPS protocol and adds

[1] The name "Twinkle" comes from observing the network: scheduling avoids collisions and thus the network twinkles if you turn on an LED every time a node transmits.

broadcast capability to enable communication from the gateway to the network while running the FPS protocol.

In this section we give a general description of the FPS protocol with an overview of the new broadcast support to provide context for the studies that follow. The focus and scope of this paper is to provide real-world experiences and evaluations of FPS with TinyOS applications and their platforms.

2.1 Power Scheduling

Power scheduling is primarily useful for low-bandwidth long-lived data-gathering applications such as GDI and TinyDB. The FPS scheme exploits the structure of a tree to build the schedule, which makes it useful primarily for data collection applications, rather than those with any-to-any communication patterns. A large class of TinyOS applications fit this model, including equipment tracking, building-wide energy monitoring, habitat monitoring [31, 29], conference-room reservations [5], art museum monitoring [26], and automatic lawn sprinklers [8]. The basic approach is to use a schedule that tells every node when to listen and when to transmit. As the bandwidth needs are low, most nodes are idle most of the time, and the radio can be turned off during these periods.

FPS scheduling is receiver initiated. In particular, the schedule spreads from the root of the tree down to the leaves based on the required bandwidth: parents advertise available slots and children that need more bandwidth request a slot. Applied recursively, this allows bandwidth allocation for all of the nodes in the network. Although this schedule ensures that parents and their children are contention free, there may still be contention due to other nodes in the network or poor time synchronization; however, this contention is rare and can be handled by a normal CSMA MAC layer.

FPS reservations correspond to a unit flow from source-node to root, and thus the schedule is really a schedule of flows. Scheduling flows reduces contention and increases fairness, and form one reason why higher-level scheduling has more value than traditional TDMA.To allow adaptive schedules, advertising continues after the initial schedule is built. If new nodes arrive, or bandwidth demands change, children can request more bandwidth or release some.

2.2 Making Reservations

Time is divided into cycles and cycles are divided into slots. Each node maintains a local schedule that indicates in what slot it transmitts, receives, or idles.The main operation is as follows:

1. Parent selects an idle slot *S* and advertises the slot
2. Child hears the advertisement and sends a request for slot *S*
3. Parent receives the request and sends an acknowledgement.

Here the parent node is the route-through node, closest to the base station. In Step 1, the parent node selects an idle slot *S* at random from its list of idle slots and advertises slot *S* during slot *C* (a specific slot known to its children). In Step 2, a child hears the advertisement and subsequently sends a request for slot *S* during slot *S*. In Step 3, the parent hears and acknowledges requests during time slot *S*. Thereafter the child transmits during slot *S* and the parent receives during slot *S*. The reservation does not need to be renegotiated and remains in effect until the child cancels the reservation or the parent times out the reservation because no receptions occur after some number of cycles. No acknowledgement implies a request was denied, and the child must petition for the next advertised reservation slot. A parent may additionally advertise slots at random times i.e. not in the *C* slot.

A node keeps its radio off during idle time slots. The one exception is when a node joins the network or switches parents. In this case it must leave its radio on until it makes an initial reservation and learns the slot *C* specific to its parent. Although made locally, these reservations represent bandwidth allocation for entire traffic flows from source to sink. This is because all nodes preallocate some amount of flow in advance. Generally speaking, local nodes observe a rule that the amount of transmission slots in their schedule must be kept greater than the amount of receive slots.

2.3 Partial Flows and Broadcast

The original FPS protocol reserves entire flows from source to sink. Twinkle introduces a new reservation type called *partial flows*. A partial flow is one that terminates at some node other than the root, i.e. the reservation is not from source to sink. Partial flows can be used for various operations such as data aggregation and compression. For example, partial flows can be used to enable in-network data aggregation, in which the flow terminates at the node that does the aggregation.

Broadcast is essential for systems like TinyDB that need to inject queries or commands into the network. In Twinkle, a broadcast channel is an instance of a partial flow. In this case the partial flows are used in the reverse direction: each node reserves a partial flow with its parent that it will use as a broadcast channel for its children. Upon joining the network, each node acquires at least one partial flow reservation that terminates at its parent. This is called the *Comm* channel (slot *C*) and is used by the node as a broadcast channel for sending synchronization packets, advertisements, and forwarding messages injected from the base station. Twinkle protocol messages always include the slot number of the Comm channel (slot *C*). In this way, children nodes know in which slot to listen for broadcasts from their parent.

Twinkle maintains two forwarding queues: one used for broadcasting or forwarding commands away from the base station, and one used for forwarding packets toward the base station. When a node receives a command message it invokes the appropriate command message handler and places the message on the command queue for forwarding. The Comm channel is shared; both injected commands and time sync packets (with slot advertisements) use the same channel. The convention is if there is a command to be forwarded that is sent first followed by the time sync packet.

```
if current slot == Comm slot
   if command in command queue
       broadcast command message
   endif
   broadcast sync packet
endif
```

The GDI application in Section 3 uses the Comm channel for time sync packets and injecting commands to start and stop the experiments. The TinyDB application in Section 4 uses the Comm channel for time sync packet and injecting TinyDB queries.

3 Application: Great Duck Island

Our first target application, GDI [18, 31], is a habitat monitoring application deployed on Great Duck Island, Maine. GDI is a sense-to-gateway application that sends periodic readings to a remote base station, which then logs the data to an Internet-accessible database. The architecture is tiered, consisting of two sensor patches, a transit network, and a remote base station. The transit network consists of three gateways and connects the two sensor patches to the remote base station. There are two classes of `mica2dot` hardware: the *burrow* mote and the *weather* mote. The burrow motes monitor the occupancy of birds in their underground burrows and the weather motes monitor the climate above the ground surface. In this section, we will draw on information about the *weather* motes provided by the study of the Great Duck Island deployment [31].

Of the two weather mote sensor patches, one is a singlehop network and the other is a multihop network. The singlehop patch is deployed in an ellipse of length 57 meters and has 21 weather motes. Data is sampled and sent every 5 minutes. The multihop network is deployed in a 221 x 71 meter area and has 36 weather motes. Data is sampled and sent every 20 minutes.

In this section we compare the end-to-end packet reception, or *yield*, and power consumption of Twinkle/FPS with the low-power listening technique [12] used at Great Duck Island. Both schemes will be running the GDI application on a 30 node laboratory testbed. We will additionally investigate the phenomena of *overhearing* in the low-power listening case.

3.1 GDI with Low-Power Listening

The GDI application uses low-power listening to reduce radio power consumption. In low-power listening, the radio periodically samples the wireless channel for incoming packets. If there is nothing to receive at each sample, the radio powers off, otherwise it wakes up from low-power listening mode to receive the incoming packet. Messages include very long preambles, so they are at least as long as the radio channel sampling interval. The advantages of low-power listening are that it reduces the cost of idle listening, integrates easily, and is complementary with other protocols. It is characterized by high end-to-end packet reception, or *yield*. This is due to the long packet preamble acting as an in-band busy-tone.

Density and multihop also impact power consumption. The GDI study [31] reports a much higher power consumption in the multihop patch than the single hop patch which resulted in a shortened network lifetime — 63 of the 90 expected days — for the multihop patch. Two causes are attributed. First, messages have a higher transmission and reception cost due to their long preambles. Second, nodes wake up from low-power listening mode not only to receive their own packets, but anytime a packet is heard, regardless of the destination. *Overhearing* is the main contributor to the higher power consumption in the multihop patch.

We also observe that although low-power listening reduces the *cost* of idle listening it does not reduce the *amount* of idle listening, so that at very low data-sampling intervals its advantage declines because the radio must continue to turn on to check for incoming packets although there are none to receive. For very low data rates, we will show that scheduling such as Twinkle becomes more attractive because the radio (and potentially other subsystems) can be deterministically powered down until it is time to be used.

3.2 GDI with Twinkle

We implemented a version of GDI in TinyOS that uses Twinkle for its radio power management. This was a rather straight forward integration that consisted of wiring the GDI application component to the Twinkle component and disabling low-power listening. The Vanderbilt TimeSync, SysTime, and SysAlarm [19] components are used for time synchronization and timers. At the time of this work, TimeSync only supported the use of SysTime, which uses the CPU clock. The implication being, that for these experiments, GDI was not able to power manage the CPU. In all of our data presented here, we subtracted the draw of the CPU as if we had used a low-power Timer implementation. A version of TimeSync using the external crystal will become available shortly.

3.3 GDI Experiments

We conducted a total of 12 experiments on two versions of the GDI application. GDI-lpl uses low-power listening for radio power management and GDI-Twinkle uses Twinkle for radio power management The experiments were run on a 30-node in-lab multihop sensor network of `mica2dot` motes.

Twinkle supports data-gathering type applications like GDI where the majority of traffic is assumed to be low-rate, periodic, and traveling toward a base station. We ran a simple routing tree algorithm provided by Twinkle based on grid locations to obtain a realistic multihop tree topology and then used the same tree topology for the 12 experiments. As is done in the Great Duck Island deployment, no retransmissions are used in these experiments.

In each experiment we varied the data sample rate: 30 seconds, 1 minute, 5 minute, and 20 minutes. For experiments with 30 second and 1 minute sample rates, 100 messages per node were transmitted. For experiments with 5 minute and 20 minute sample rates, 48 and 12 messages were transmitted per node respectively. In the GDI-lpl experiments we varied the channel sampling interval: 485 ms and 100 ms. All experiments collected node id, sequence number, routing tree parent, routing tree depth, node temperature, and node voltage. The GDI-Twinkle experiments additionally collected the number of children, number of reserved slots, current transmission slot, current cycle, and number of radio-on slots per sample period.

3.4 Measuring Power Consumption

During the experiments, we measure the power consumption directly, using an oscilloscope, of two nodes located in two separate places of interest in the network. One node, we call the *inner node*, is located one hop from the base station and has a heavy amount of route-through traffic that is similar to its routing one-hop siblings. This should give us an estimate of the maximum lifetime of the network. This is a common method, documented by several researchers, for example [29]. In addition we measure the current at a second node. The second node is a *leaf node* that is one-hop from the base station as well. As it does not route-through any traffic, we should be able to see the effect of overhearing on power consumption at a node in a busy part of the network. If the measured current of the *inner node* and *leaf node* are similar in their active cycles, then we know the inner node is experiencing overhearing since all other factors remain the same. This is an important aspect of evaluating low-power listening.

At the lower sample rates, it is not feasible to take a measurement over the entire sample period, so we design our experiments so that we take some direct measurements and extrapolate others. For GDI-Twinkle, we define a cycle to be 30 seconds. Thus, full sample periods for the 30-second, 1-minute, 5-minute, and 20-minute sample rates are 1, 2, 10, and 40 cycles respectively. We schedule all data traffic during one cycle of each sample period called the *active cycle*. The unscheduled cycles are called *passive cycles*. Both active and passive cycles include protocol traffic (i.e. sending advertisements and listening for requests). We then measure the current at the two motes capturing data from both active and passive cycles during the 1 minute sample rate experiment. Then we take a running windowed average over a full 1-minute period, which gives us the power draw for both an active and passive cycle. Table 1 presents these direct power measurements.

For GDI-lpl we follow a similar method. We measure current at the two motes capturing data from both active and passive periods during the 1-minute sample period experiment. To represent an active period, we take a running average over the full 1-minute period. This also captures all the overhearing that occurs at the mote during a full period of any given sample rate. To represent a passive period, we took the longest chain of data from the measurements in which only idle channel sampling occurred. From this information we calculate the power consumption for the 5-minute and 20-

Table 1. Power Measurement (mW)

Power Management	Period (Sec)	Inner (mW)	Leaf (mW)
Twinkle active	30	2.18	0.69
Twinkle passive	30	0.33	0.33
Lpl-485 active	60	16.5	16.0
Lpl-485 passive	60	0.99	0.99
Lpl-100 active	60	8.20	7.60
Lpl-100 passive	60	3.90	3.90

minute sample rate experiments. The 30-second sample rate was measured separately (not calculated) and is shown in Table 1.

3.5 Evaluation

In this section we discuss the results of the data from all 12 experiments, and we also compare with the actual GDI deployment data.

Power Comparison with Low-Power Listening. Given the direct power measurements from Table 1, we can estimate the power consumption for the 5-minute and 20-minute sample rate experiments. For example, for Twinkle, we read off the following: an active cycle at the inner mote consumes 2.18 mW and a passive cycle consumes 0.33 mW. Given these numbers, for a 20-minute sampling rate we expect 1 active cycle and 39 passive cycles, for a weighted average of 0.38 mW. For the leaf mote, an active cycle consumes 0.69 mW and a passive cycle consumes 0.33 mW, giving a weighted average of 0.34 mW.

Similarly, to compute the GDI-lpl power consumption at a 20-minute sample rate we assume that for one minute the application consumes the energy of the active period and for the remaining 19 minutes the application consumes the energy of the passive period. Using the values from Table 1, the inner mote during the 20-minute sample rate Lpl-100 experiment, would consume an average of 4.12 mW ((8.2+19*3.9)/20 = 4.12mW).

Figure 1 shows all four sample periods: the 30-second and 1-minute rates are measured, and the 5-minute and 20-minute periods are estimated as above. For Twinkle, the inner node consistently has a greater draw than the leaf node. In contrast, for LPL, the inner and leaf nodes consistently have almost the same draw. This indicates that Twinkle's main power draw depends on the routed traffic, and in most of the cases LPL's main power draw depends on the overheard traffic. However, from Table 1 we see the passive power draw for LPL-100 is 3.9 mW, which forms an asymptote as the

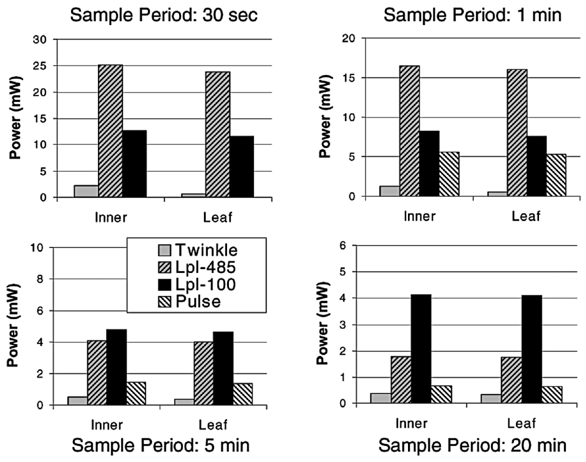

Fig. 1. Relative power consumption of Twinkle and LPL for four different sample periods. Pulse is a newer version of LPL discussed below.

sample period increases. Overall, as the sample rate gets lower and the preambles get shorter, overhearing does not play as big role.

The next thing to notice is at the higher sample rates, LPL-485 has a higher power consumption than LPL-100, but at the lower sample rates, LPL-485 has a lower power consumption than LPL-100. This reveals a relationship within LPL where as the cost of transmitting increases with longer preambles, the cost of channel sampling decreases with longer sampling intervals.

Finally, we added a newer variation of LPL to the figure, called Pulse. Pulse was developed as part of BMAC [20], and optimizes the power consumption of LPL by listening for energy in the channel rather than the decoded preamble. This reduces the cost of listening substantially. Because it has much stricter timing constraints, Pulse cannot run on the `mica2dot` platform. However, we can compute the active and passive estimates for Pulse as if it were running on the `mica2dot` given our power traces and Table 2 from the BMAC paper, which provides the raw listening cost. Although Pulse does perform better than LPL, it is still 2x to 5x higher power consumption than Twinkle. Across the board, Twinkle has better power consumption than LPL, with improvements that range from 2x (over Pulse for low rates) to 10x (in cases where the listening interval is poorly chosen).

Table 2. Yield and Fairness Comparison

Power Scheme	Sample Period	Yield	Max/ Min
Twinkle	0.5	0.80	2.11
Twinkle	1	0.90	1.74
Twinkle	5	0.84	1.92
Twinkle	20	0.83	2.4
Lpl-485	0.5	0.40	15.6
Lpl-485	1	0.68	94.0
Lpl-485	5	0.72	11.8
Lpl-485	20	0.69	12.0
Lpl-100	0.5	0.85	3.45
Lpl-100	1	0.83	2.23
Lpl-100	5	0.78	2.76
Lpl-100	20	0.77	4.00

Yield and Fairness. Table 2 shows the average yield (end-to-end packet reception) for all 12 experiments, and the ratio of the best and worst throughputs (Max/Min). This ratio indicates fairness: lower ratios are more fair. At 30 seconds, the LPL-485 network is saturated due to the long preambles and this accounts for its low yield. Overall, both Twinkle and LPL-100 are significantly better than LPL-485. Twinkle shows better fairness than LPL-100 and, other than the 30 second sample rate, Twinkle has higher yield than LPL-100.

Understanding the GDI Field Study. Viewing the data in comparison to the data provided by the GDI study [31], we find the results in the laboratory are remarkably close to the results in the field.The Great Duck Island deployment used a low-power listening channel sampling interval of 485 ms, a data sample period of 20 minutes in the multihop patch, and a data sample period of 5 minutes in the singlehop patch.

Table 3 presents results taken from the GDI field study, labeled GDI-485, and ¡ncludes data from four of our in-lab experiments, labeled LPL-485 and Twinkle. For each row, we report the sample period, average yield, inner and leaf power consumption, and the number of nodes in the experiment. *For GDI-485, the yield figure represents the average yield from the first day of deployment.*

A close comparison can be drawn between LPL-485 and GDI-485 at the 20 minute sample rate. LPL-485 has a power draw of ~1.76 mW while GDI-485 has a power draw of 1.6 mW. The GDI-485 figure is expected to be lower for two reasons: in the laboratory, the two measured nodes are from the busier section of the testbed, and the testbed has a constant load rather than a decreasing one. In the GDI deployment, some multihop motes died and stopped sourcing traffic, which is why we report yield from the first day of deployment.

The yield data is extremely close as well. All yields for LPL-485 and GDI-485 are ~70%. The only large difference between the two data sets is the power consumption

Table 3. Comparison of our lab data with the actual GDI field study

Power Mgnt	**Sample Period**	**Yield**	**Inner (mW)**	**Leaf (mW)**	#
GDI-485 (single)	5	0.70	n/a	0.71	21
GDI-485 (multi)	20	0.70	1.60	n/a	36
Lpl-485	5	0.72	4.09	3.99	30
Lpl-485	20	0.69	1.77	1.74	30
Twinkle	5	0.84	0.52	0.36	30
Twinkle	20	0.83	0.38	0.34	30

at the 5-minute sample period. This is easily explained by recalling that at the 5-minute sample period, GDI-485 is singlehop while LPL-485 is multihop, and the LPL-485 measurements include a large amount of overhearing.

The closeness of the LPL-485 and GDI-485 data gives us high confidence in the corrrectness of our methodology and the results of our laboratory experiments. We expect the Twinkle numbers are a good estimate of how Twinkle would do were we to have access to a field deployment. Our laboratory experiments show that Twinkle consumes at least 4x less power and provides about 14% better yield.

4 Application: Redwoods with TinyDB

Our second target application, TinyDB [17], is a distributed query processor for TinyOS motes. TinyDB consists of a declarative SQL-like query language, a virtual database table, and a Java API for issuing queries and collecting results. Conceptually the entire network is viewed as a single table called *sensors* where the attributes are inputs of the motes (e.g. temperature, light) and queries are issued against the *sensors* table. The SQL language is extended to include an "`EPOCH DURATION`" clause that specifies the sample rate.

A typical query looks like this:

```
SELECT nodeid, temperature
FROM sensors
EPOCH DURATION 3 min
```

TinyDB allows up to two queries running concurrently: one for sensor readings and one for network monitoring. In this section we compare the power savings of TinyDB using Twinkle versus TinyDB using application-level duty cycling — the power management scheme currently used in TinyDB. We estimate the power savings of the two approaches using the TinyDB Redwood deployment in the Berkeley Botanical Garden [14] as our topology and traffic model.

4.1 Estimating Power Consumption

Determing the power consumption of TinyDB with application-level duty cycling is straight forward. For this analysis we will estimate the power consumption of both the `mica` and `mica2` platforms and take an in-depth look at a radio trace generated by TinyDB with Twinkle. We use the following three-part methodology:

1. Estimate the amount of time the radio is on and off for each scheme. Our metric for this will be radio on time per hour, measured in seconds.
2. For Twinkle, we validate this estimate in Section 4.5 by looking in detail at one of the motes. The radio on time for application-level duty cycling is easy to estimate.
3. We use actual measured current we obtained from `mica` and `mica2` motes using an oscilloscope (Table 6) to estimate power consumption for radio on/off times. (In the GDI application we measured the current directly during the experiment.)

Listening for information on the radio channel is of a cost similar to transmission of data [23,24,4], so this combination provides a reasonably accurate overall view of power consumption, which although not perfect, is certainly very accurate relative to the 4.3X advantage in power shown by Twinkle in Section 4.6.

4.2 Topology and Traffic Model

The Redwood deployment has 35 mica2dot motes dispersed across two trees reporting to one base station in the Berkeley Botanical Gardens. Each tree has 3 tiers of 5 nodes each and 2 nodes placed at each crest. One tree has 1 additional node at a bottom branch. Every 2.5 minutes each mote transmits its query results, which are multi-hopped and logged at the base station.

By examining the records in the redwood database, we can derive the actual topology information, and from this construct a general topology that reflects its state the majority of the time.

Out of 35 nodes, generally 2/3 of the nodes are one hop from the base station and 1/3 of the nodes are two hops from the base station at any given time. We start by computing the radio on time per hour for the case with no power management:

60 sec/min * 60 min/hour = 3600 sec/hour
No power management = 3600 sec/hour

This number is the average amount of time each radio is on per hour throughout the deployment. We next estimate this metric for duty cycling followed by an estimate for FPS.

4.3 Duty Cycling

In TinyDB duty cycling, the default power management scheme, all nodes wake up at the same time for a fixed waking period every EPOCH. During the waking period nodes exchange messages and take sensor readings. Outside the waking period the processor, radio, and sensors are powered down. Estimating the radio-on time is thus straightforward: all 35 nodes wake up at the same time every 2.5 minutes for 4 seconds and exchange messages. The sample rate is thus 24 samples per hour. Each node is on for 96 sec/hour.

24 samples/hour * 4 sec/sample = 96 sec/hour
Duty Cycling = 96 sec/hour

As expected, this approach is subject to very high packet losses due to the contention produced by exchanging packets at nearly the same time. A recent TinyDB empirical study [29] shows high losses, between 43% and 50%, and high variance using duty cycling. Although we do not test it explicitly, there is no reason to expect the yield for Twinkle (or low-power listening) would deviate from the 80% shown in Section 3.

4.4 Twinkle

Topology, time-slot duration, protocol traffic, and data traffic are factors in estimating the radio-on time for Twinkle. We will use the same topology as above for estimating

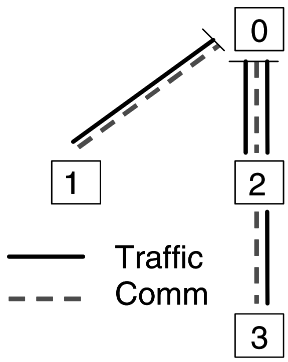

Fig. 2. Topology and Traffic for Estimates

the radio-on time of the 35 nodes. Time-slot duration and number of slots per cycle are configuration parameters in Twinkle. For this example, the time slot duration is 128 ms and for simplicity, the number of slots per cycle is 1172, which is roughly 2.5 minutes. Because of the long cycle length, we will add an extra advertisement per cycle.

Figure 2 depicts our subtree topology and traffic model. Solid lines represent data traffic (T/R) that is forwarded from the network to the base station every cycle. Dashed lines represent a Broadcast channel used for protocol traffic (B/RB). The Broadcast channel is used for TinyDB queries, network protocol messages, and advertisements.

Given the topology in Figure 2 and traffic in Table 4 we can now calculate the radio-on time for each node. Node 0 is the base station and has no cost. There is a cost of 3 time slots for advertisements (A): one advertisement, one receive pending, and one receive pending for the advertisement sent during the Broadcast.

Thus, this model captures data traffic as well as protocol traffic (i.e. sending advertisements and listening for requests).

Table 4. Traffic per Cycle (number of time slots)

Node	**T**	**R**	**B**	**RB**	**A**
1	1	0	1	0	3
2	2	1	1	1	3
3	1	0	1	0	3

For each node the cost is 0.767 seconds per cycle:

```
5(T/R) + 4(B/RB)+ 9(A)
   = 18 * 128ms
   = 2.3 sec/cycle per 3 nodes
   = 0.767 sec/cycle (per node)
```

At 24 samples per hour, on average, each node is on 18.4 sec/hour:

```
24 samples/hour * 0.767 sec/cycle
   = 18.4 sec/hour
```

Twinkle = 18.4sec/hour

This is a savings of 5.2x compared with the duty cycle approach and 196x compared with no power management. In addition, the radio-on time is actually overestimated. Transmit slots do not leave the radio on for the whole slot since they can stop once their message is sent; this is shown in detail in the next section.

4.5 Twinkle Validation and Radio Trace

We implemented a prototype of TinyDB that uses Twinkle for radio power management. To validate our prototype and the radio on/off times, we ran the following experiment on three `mica2dot` motes and one `mica2` mote as base station arranged in a topology shown in Figure 2. We monitored intermediate node 2 while it forwarded packets and sent advertisements. There are 64 slots of 128 ms each per cycle. We instrumented TinyDB-Twinkle to record the time of each call to turn the radio on and radio off, the beginning time of each time slot, and the state of each slot.

From the TinyDB Java tool we issue the query:

```
SELECT nodeid
FROM sensors
EPOCH DURATION 8192 ms
```

The intermediate mote is connected to an Ethernet device, and the debug records are logged over the network to a file on the PC. The regular query results are multi-hopped to the base station and displayed by the Java tool. In this experiment, we expect to have 1 advertisement, 2 receive pending slots, 3 transmit slots (one is a broadcast), 2 receive slots, and 56 idle slots per 64-slot cycle. We validate both the count of idle slots against the radio off time shown in Table 5.

Table 5. Validating Idle Slots

Metric	Slots	Idle %
Predicted Idle Slots	56/64	89.1%
Measured Idle Slots	56/64	89.1%
Measured Radio Off Time	—	91%

Note that the radio off time is higher than the percentage of idle slots because Transmit slots turn the radio off early — as soon as their messages have been sent.

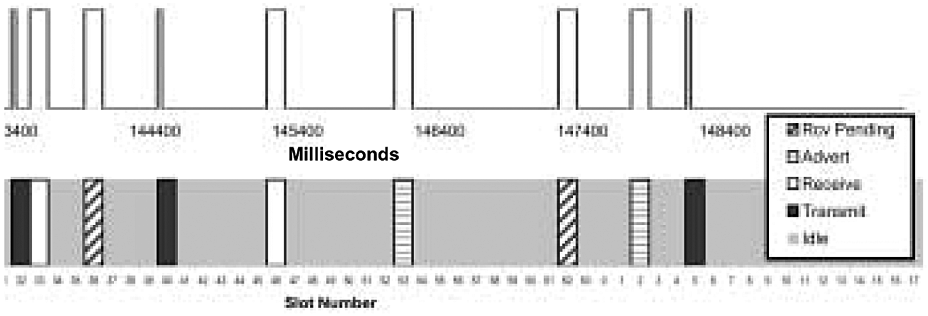

Fig. 3. A subsection of the validation experiment. The top graph shows actual radio on/ off times in milliseconds. The bottom graph shows the measured Twinkle state versus slot numbers aligned with time Note that the radio is always off for Idle cycles and that for Transmit cycles the on time is just long enough to transmit the queued messages.

Figure 3 shows a subsection of the validation experiment. The top graph shows actual radio on/off times (milliseconds). The bottom graph shows the measured Twinkle state versus slot numbers aligned with time; this subsection shows the active portion of a cycle (slots not shown are idle). Note that the radio is always off for Idle slots and that for Transmit slots the on time is just long enough to transmit the queued messages. In this experiment, the time slot duration is 128 ms, there are 64 slots per cycle, and the advertising frequency is once per cycle. This cut shows two advertisement slots, which is fine given that they are actually in two different cycles.

This experiment validates our methodology and shows that the power estimate for Twinkle in the previous section is actually conservative (since we count all of the Transmit slot time).

4.6 Power Savings

Finally, given the validated radio on times, we can estimate the power savings. First, however we need to know the current draw for a mote depending on whether or not the radio is on, and/or the CPU is on. With an oscilloscope, we measured the current of the `mica` and `mica2` motes in three states: asleep, cpu idle, and both cpu and radio on. The results are shown in Table 6.[2]

Given these current draws, we estimate power consumption as:

```
Power (mAh) =
(On time)*(On draw) + (Off time)*(Off draw)
```

Using this equation and the radio-on times summarized in Table 7, we estimate the power consumption depicted in Figure 4. In all cases, both Duty Cycling and Twinkle perform substantially better (lower power) than no power management, so we focus on the difference between Twinkle and Duty Cycling.

[2] `Mica2` radio power varies from 7.4 to 15.8 mA depending on transmit power, plus 7.8 mA for the *active* CPU draw for a total of 15.2 to 23.6 mA. We use 20mA as an overall estimate.

Table 6. Power Consumption of Motes (mA)

Scheme	Radio On Time	Ratio
None	3600	196
Duty Cycling	96	5.2
Twinkle	18.4	1

Table 7. On Times (seconds per hour)

Mote	Asleep	CPU Idle	CPU+Radio On
Mica1	0.01	0.4	8.0
Mica2	0.03	3.9	20

The biggest issue for estimating the power savings is whether or not the CPU is asleep when the radio is off. Neither system needs the CPU per se during idle times, but some sensors may require CPU power. Thus we expect for both the `mica1` and `mica2` the "CPU asleep" numbers are more realistic and we will quote these in our overall conclusions. However, we include the "CPU on" case for completeness. Note that even for cases where the CPU is needed for sensor sampling, the "CPU asleep" graph is more accurate, since the CPU would be asleep most of the time.

For the CPU on case, Twinkle outperforms Duty Cycling by 37% on the `mica1` and 8% on the `mica2`, which has a higher CPU current draw. Compared to no power management, the advantage for Twinkle is 18X and 5X respectively.

For the more realistic "CPU asleep" case, i.e. the CPU is asleep during Idle slots, Twinkle outperforms Duty Cycling by 4.4X on the `Mica1` and 4.3X on the `Mica2`. Note that this is consistent with the 5.2X reduction in radio on time. Compared to no power management, the advantage for Twinkle is 160X and 150X respectively.

Thus to summarize, for the TinyDB application with the Redwood study workload, we see a power savings of about 4.3X over Duty Cycling and 150X over no power management.

5 Related Work

Power consumption is an important issue in wireless sensor networks and energy optimizations are considered at all layers of the hardware and software platform. Many researchers have investigated energy efficient protocols in software to reduce communication costs.

In the area of energy-efficient MAC layers, there are two broad classes of approaches: contention based [22,33,7] and TDMA based [28,1,6]. PAMAS [22] enhances the MACA protocol with the addition of a signaling channel. It powers down the radio when it hears transmissions over the data channel or receptions over the signaling channel. S-MAC [33] incorporates periodic listen/sleep windows of fixed sizes similar to 802.11 PS mode [16]. In order to communicate, neighboring nodes periodically exchange their listen schedules. In the listen phase nodes transmit RTS/CTS packets and in the sleep phase nodes either transmit data or sleep if there is no data to

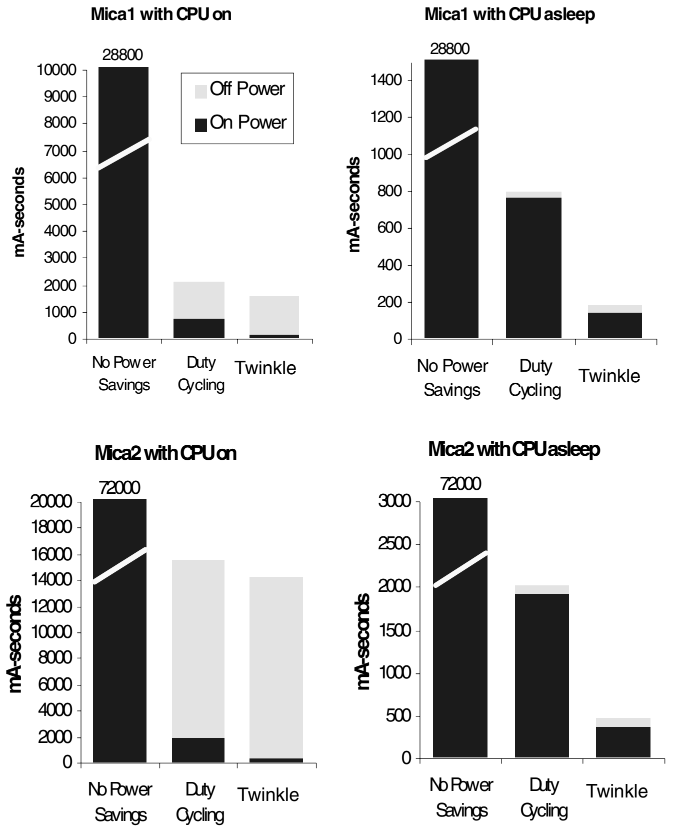

Fig. 4. Estimating power savings for two families of motes (Mica1, top, and Mica2, bottom), with the CPU on or asleep when the radio is off. Each vertical axis has a different scale, and in all cases the "No power savings" column goes off the top (Mica1 28800, Mica2 72000 mA-secsonds). Light gray is the radio-off power consumed (per hour), while dark gray is the radio-on power consumed.

send. T-MAC [7] is a variation on S-MAC. Instead of using a fixed listen window size, it transmits all messages in bursts of variable length, and sleeps between bursts.

TDMA-based protocols have natural idle times built into their schedules where the radio can be powered down. Additionally they do not have to keep the radio on to detect contention and avoid collisions. Centralized energy management [1] uses cluster-heads to manage CPU and radio consumption within a cluster. Centralized solutions generally do not scale well because inter-cluster communication and interference is hard to manage. Self organization [28] does not use clusters or hierarchies. It has a notion of super frames similar to TDMA frames for time schedules and requires a radio with multiple frequencies. It assumes a stationary network and generates static schedules. This scheme has less than optimal bandwidth allocation. Slot reservations can only be used by the node that has the reservation. Other nodes cannot reuse the slot reservation.

ReOrgReSync[6] uses a combination of topology management (ReOrg) and channel access (ReSync) and relies on a backbone for connectivity. Relay Organization is a topology management protocol which systematically shifts the network's routing burden to energy-rich nodes (wall powered and battery powered nodes). Relay Synchronization (ReSync), is a TDMA-like protocol that divides time into epochs. Nodes periodically broadcast small intent messages at a fixed time which indicate when they will send the next data message. All neighbors listen during each others intent message times. It assumes a low data rate and only one message per epoch can be sent.

Energy-efficient routing in wireless ad-hoc networks has been explored by many authors, see [25,34,15,10] for examples. Topology management approaches exploit redundancy to conserve energy in high-density networks. Redundant nodes from a routing perspective are detected and deactivated. Examples of these approaches are GAF [32] and SPAN [3]. Our approach does not seek to find minimum routes or redundancy. These protocols are designed for systems that require much more general communication throughout the network.

6 Conclusion

In this paper we have presented our experiences with Twinkle, the next-generation implementation of FPS, and evaluated its use for two real-world TinyOS applications and three mote platforms. We demonstrated that Twinkle can save 2-5x of the power consumption for real applications that already use power management of some kind. We saw a 2-4x improvement for the GDI application, and about 4x for the TinyDB Redwoods deployment. We also covered an important enhancement to the idea of network-layer power scheduling — the concept of scheduling partial flows that enable broadcast — to make network power scheduling a realistic alternative for real deployments of TinyOS applications.

Acknowledgments. We are much indebted to several individuals of the TinyOS community for their collaborations, suggestions, and support. Rob Szewczyk for his continuous help and advice on power management, TinyOS, and GDI. Brano Kusy and Miklos Maroti for their support on timers and time synchronization. Sam Madden for his support on TinyDB and Gilman Tolle for his work on the Redwoods database. This work was supported, in part, by the Defence Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grant NSF IIS-033017), and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant IEA-9802069).

References

[1] K.A. Arisha, M.A. Youssef, M.F. Younis,"Energy-aware TDMA based MAC for sensor networks," IEEE IMPACCT 2002, New York City, NY, USA, May 2002.

[2] G. Asada, M. Dong, T. S. Lin, F. Newberg, G. Pottie, W. J. Kaiser, H. O. Marcy,"Wireless integrated network sensors: low power systems on a chip," ESSCIRC '98. Proceedings of the 24th European Solid-State Circuits Conference, The Hague, Netherlands, September 1998.

[3] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, "Span: an energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," MobiCom 2001, Rome Italy, July 2001.
[4] Chipcon. http://www.chipcon.com/files/CC1000_Data_Sheet_2_3.pdf
[5] W.S. Conner, L. Krishnamurthy, and R. Want, "Making everyday life a little easier using dense sensor networks," Proceeding of ACM Ubicomp 2001, Atlanta, GA, Oct. 2001.
[6] W.S. Conner, J.Chhabra, M. Yarvis, L.Krishnamurthy, "Experimental Evaluation of Topology Control and Synchronization for In-building Sensor Network Applications," ACM Workshop on Wireless Sensor Networks and Applications, September 2003.
[7] T.van Dam, K. Langendoen, "An Adaptive Energy-Efficient MAC Protocol for Wireless Sensor Networks," SENSYS 2003, Los Angeles, CA, USA, November 2003.
[8] Digital Sun, Inc.: http://digitalsun.com
[9] L. Doherty, B.A. Warneke, B.E. Boser, K.S.J. Pister, "Energy and Performance Considerations for Smart Dust," International Journal of Parallel Distributed Systems and Networks, Volume 4, Number 3, 2001, pp. 121-133.
[10] Z. Haas, J. Halpern, and L. Li, "Gossip-based ad-hoc routing," IEEE INFOCOM 2002, New York, NY, USA, June 2002.
[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K.S.J. Pister, "System architecture directions for networked sensors," ASPLOS 2000, Cambridge, MA, USA, November 2000.
[12] J. Hill, D. Culler, "Mica: a wireless platform for deeply embedded networks," IEEE Micro, 22(6):12-24, November/December 2002.
[13] B. Hohlt, L. Doherty, E. Brewer, "Flexible Power Scheduling for Sensor Networks," IPSN 2004, Berkeley, CA, USA, April 2004.
[14] W. Hong, "TASK In Redwood Trees", http://today.cs.berkeley.edu/retreat-1-04/weihong-task-redwood-talk.pdf, NEST Retreat, Jan 2004.
[15] B. Karp and H.T. Kung, "GPSR: Greedy Perimeter Stateless Routing for wireless networks," MobiCom 2000, Boston, MA, USA, August 2000.
[16] LAN MAN Standards Committee of the IEEE Computer Society, "IEEE Standard 802.11, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications," IEEE, August 1999.
[17] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TAG: a tiny aggregation service for ad-hoc sensor networks," 5th Symposium on Operating Systems Design and Implementation, Boston, MA, USA, December 2002.
[18] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, J. Anderson, "Wireless sensor networks for habitat monitoring," WSNA 2002, Atlanta, GA, USA, September 2002.
[19] M. Maroti, B. Kusy, G. Simon, A. Ledeczi, "The Flooding Time Synchronization Protocol," SenSys 2004, Baltimore, MD, USA, November 2004.
[20] J.Polastre, J.Hill, D.Culler, "Versatile Low Power Media Access for Wireless Sensor Networks", SenSys 2004, Baltimore, ML, USA.
[21] G.J. Pottie, W.J. Kaiser, "Wireless Integrated Network Sensors," Communications of the ACM, vol. 4, no. 5, May 2000.
[22] C.S. Raghavendra and S. Singh, "PAMAS - Power aware multi-access protocol with signaling for ad hoc networks," ACM Communications Review, vol. 28, no. 33, July 1998.
[23] V. Raghunathan, C. Schurgers, S. Park, and M.B. Srivastava, "Energy-aware wireless microsensor networks," IEEE Signal Processing Magazine, vol. 19, no. 2, March 2002.
[24] RFM Monolithics. http://www.rfm.com/products/data/tr1000.pdf.
[25] E. M. Royer and C-K. Toh. "A review of current routing protocols for ad-hoc mobile wireless networks," IEEE Personal Communications, April 1999.

[26] Sensicast Systems: http://www.sensicast.com.

[27] K. Sohrabi, J. Gao, V. Ailawadhi, and G.J. Pottie, "Protocols for self-organization of a wireless sensor network," IEEE Personal Communications, Oct. 2000.

[28] K. Sohrabi and G.J. Pottie, "Performance of a novel self-organization for wireless ad-hoc sensor networks," IEEE Vehicular Technology Conference, 1999, Houston, TX, May 1999.

[29] P. Buonadonna, J. Hellerstein, W. Hong, D. Gay, S. Madden, "TASK: Sensor Network in a Box", European Workshop on Wireless Sensor Networks 2005, Istanbul, Turkey, February 2005.

[30] M. Stemm and R. Katz, "Measuring and reducing energy consumption of network interfaces in hand-held devices," IEICE Trans. on Communications, vol. E80-B, no. 8, pp. 1125-1131, August 1997.

[31] R.Szewczyk, A.Mainwaring, J.Polastre, J.Anderson, D.Culler, "An Analysis of a Large Scale Habitat Monitoring Application", SenSys 2004, Baltimore, ML, USA, November 2004.

[32] Y. Xu, J. Heidemann, D. Estrin, "Geography-informed energy conservation for ad hoc routing," MobiCom 2001, Rome, Italy, July 2001.

[33] W. Ye, J. Heidemann, D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," IEEE INFOCOM 2002, New York City, NY, USA, June 2002.

[34] Y. Yu, R. Govindan, and D. Estrin. "Geographical and Energy Aware Routing: a recursive data dissemination protocol for wireless sensor networks," UCLA Computer Science Department Technical Report UCLA/CSD-TR-01-0023, May 2001.