Link Layer Driver Architecture for Unified Radio Power Management in Wireless Sensor Networks

Kevin Klues
UC Berkeley
Berkeley, California 94720
klueska@eecs.berkeley.edu

Guoliang Xing
Michigan State University
East Lansing, MI 48824
glxing@msu.edu

Chenyang Lu
Washington University in St. Louis
St. Louis, MO 63130
lu@cse.wustl.edu

Wireless sensor networks (WSNs) represent a new generation of networked embedded systems that must achieve long lifetimes on scarce amounts of energy. Since radio communication accounts for the primary source of power drain in these networks, a large number of different radio power management protocols have been proposed. However, the lack of operating system support for flexibly integrating them with a diverse set of applications and network platforms has made them difficult to use. This paper focuses on providing link layer support towards realizing a *Unified Power Management Architecture (UPMA)* for WSNs. In contrast to existing monolithic approaches, we provide (1) a set of standard interfaces that separate link layer power management protocols from common MAC level functionality, (2) an architectural framework that allows applications to easily swap out different power management protocols depending on its needs, and (3) a mechanism for coordinating multiple applications with different power management requirements. We have implemented our approach on both the Mica2 and Telosb radio drivers in TinyOS-2.0, the second generation of the de facto standard operating system for WSNs. Microbenchmark results show that our approach can coordinate the power management requirements of multiple applications in a platform independent fashion while incurring negligible overhead.

Categories and Subject Descriptors: C.2.2 [**Computer Communication Networks**]: Network Protocols—*Protocol Architecture*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*

General Terms: Experimentation, Design

Additional Key Words and Phrases: Wireless Sensor Networks, Radio Power Management, Architecture, Framework

1. INTRODUCTION

Wireless sensor networks are a unique type of networked embedded system in which ultra low power operation is crucial. Wireless sensor networks typically consist of a large number of sensor nodes. In many applications, these nodes must be able run

unattended for months or years without any human interaction, making it essential to save energy wherever possible.

Wireless sensor network applications can range from low data rate habitat monitoring applications [Szewczyk et al. 2004][Tolle et al. 2005] to higher data rate intruder detection and tracking applications [He et al. 2006]. While energy is always a major concern, different application requirements can influence the way in which energy is to be preserved. Habitat monitoring applications, for example, have steady, periodic traffic allowing them to duty cycle their radios in order to reduce energy consumption. Latencies in message delivery are tolerable, and sometimes even encouraged if it means that better energy savings can be achieved. Applications such as intruder detection and tracking, on the other hand, rely on fast message delivery due to the urgency of the information they are providing. These applications are dominated by long periods of inactivity, followed by large bursts of traffic once something has been detected. They would rather sacrifice some energy savings for an increased level of performance once they start generating messages.

As evidenced above, it is not always possible to use the same energy management protocol for every type of application. A protocol optimized for low data rate, periodic habitat monitoring applications may perform poorly in an intruder detection application [Ye et al. 2006], whereas a protocol designed for bursty workloads would be unnecessarily complex for a simple habitat monitoring application. There must therefore be a way to allow an application to incorporate the use of whichever power management protocol is most appropriate for it.

At times, it may also be desirable to allow multiple applications to run concurrently on a single node. Imagine a set of real-time monitoring applications that each take readings from different sensors at different sampling rates. If each of these applications were to be installed on a single node, we would need a way of resolving the different requirements they impose on an underlying power management protocol. Sometimes it may be sufficient to configure a single protocol accordingly. Other times it may be more convenient to choose multiple protocols from a set of predefined ones and find a way to compose them together in some coherent way.

Unfortunately, despite significant progress in the development of power management protocols for wireless sensor networks, existing systems still lack radio driver support for flexibly choosing which ones are most appropriate and allowing them to be plugged directly into an existing system. Traditionally, MAC protocols and power management protocols have been developed together to produce a single monolithic implementation. Development of a new power management protocol meant redesigning an entire radio stack from the ground up.

To address these issues, we propose a radio driver architecture that not only allows different power management protocols to be *flexibly* integrated into a fully functional wireless sensor network system, but also allows the requirements imposed by multiple applications[1] to be *coordinated* in such a way that a single coherent power management solution is produced. With radio communication as the single largest source of energy consumption in wireless sensor networks, radio power man-

[1]The word application is overloaded. An application is anything actually sitting on top of the MAC layer, including the traditional notion of an application, as well as duty cycling protocols such as PEAS (as discussed in Section 5.2).

agement protocols are at the core of any such solution. While these radio power management protocols may exist across different layers in the networking protocol stack, this work focuses on duty cycling protocols existing only at the data link layer. A high-level overview of an overall *Unified radio Power Management Architecture (UPMA)* is presented in [Klues et al. 2007].

Specifically, this paper makes the following primary contributions: (1) We design and implement a set of interfaces that allow different radio power management protocols to share common MAC functionality at the data link layer. The separation we propose gives different applications the ability to incorporate the use of whichever power management protocol is best suited to its needs, independent of the underlying MAC protocol it relies on. [2] (2) We propose an architectural framework that gives multiple applications the ability to specify their (sometimes differing) requirements to a single underlying radio power management protocol. Coordination of these requirements is achieved through a customizable component whose implementation depends on both the power management protocol in use and the types of applications running on top of it. (3) We demonstrate the practicality of this architecture by implementing it within the TinyOS operating system on top of two platforms in wide use in wireless sensor networks today. We show that separating power management protocols from the MAC level functionality on which they rely increases flexibility while introducing only a negligible performance penalty. (4) Finally, we provide two case studies demonstrating how differing requirements from multiple applications can be resolved within our architecture. In each case study we show how to coordinate these requirements differently.

The rest of this paper is organized as follows. Section 2 first provides an overview of typical power management protocols for which our architecture is being developed. Section 3 then presents the design of our architecture with Section 4 providing example implementations demonstrating its various features. Section 5 presents experimental results of the overhead incurred by implementing power management protocols using our architecture, as well as results obtained during our two case studies. Finally, Section 6 concludes the paper.

2. POWER MANAGEMENT APPROACHES

In this section, we review existing approaches to radio power management in WSNs. This review provides the basis for our design of a common set of interfaces and a unified architecture that can support diverse power management protocols.

Existing approaches to radio power management fall into two categories: transmission power control and duty cycling. Transmission power control [Santi 2005] reduces the energy consumed during communication by adjusting the power at which a radio transmits. Duty cycling reduces energy wasted during idle listening by allowing the radio to cycle between periods of activity and sleep. The architecture presented in this paper only focuses on supporting duty cycling protocols, which have proven to be a very efficient means of extending the system lifetime

[2] Certain power management optimizations (such as overhearing avoidance) may still need to be implemented *within* the MAC layer itself, but features such as this are beneficial to all power management protocols existing at the link layer, and their existence does not diminish the need for providing the separation that we do.

of WSNs. It can be accomplished by following one of three different approaches: TDMA, scheduled contention, or channel polling [Ye et al. 2006].

In TDMA based protocols, time is divided up into discrete time slots and allocated to all nodes within transmission range of one another. Nodes transmit during the time slots that have been allocated to them, and listen during the time slots that have been allocated to those nodes from which they wish to receive. When not transmitting or receiving, a node is free to sleep. Several different TDMA based protocols have been proposed for use in WSNs. These protocols include TRAMA [Rajendran et al. 2003], and DRAND [Rhee et al. 2006]. One limitation of these types of protocols is that their schedules can be very sensitive to changes in network traffic or network topology, and all nodes sharing a schedule must remain synchronized with one another. Whenever one of these properties changes, a new TDMA schedule must somehow be generated and distributed to some subset of nodes in the network.

Protocols based on scheduled contention allow nodes to schedule times in which they will all be awake in order to communicate. Nodes within transmission range of one another synchronize their schedules to ensure that they are all awake at the same time. While awake, nodes contend for use of the radio channel through a process known as CSMA/CA. Nodes that gain access to the channel during this contention period are allowed to send, while all other nodes listen. Several energy-efficient MAC protocols based on sleep scheduling include S-MAC [Ye et al. 2004], T-MAC [van Dam and Langendoen 2003], and Z-MAC [Rhee et al. 2005]. Despite the energy savings achieved using protocols of this type, energy is still wasted in maintaining synchronization and waking up when there is no data to transmit or receive.

Protocols based on channel polling (such as the Low Power Listening features of B-MAC [Polastre et al. 2004], WiseMac [El-Hoiyi et al. 2004] and X-MAC [Buettner et al. 2006]) do not require synchronized contention periods for reception and transmission. All nodes independently wake up to poll the radio channel for activity. If there is, they prepare themselves for message reception. If there is not, they return immediately to sleep. Transmitting nodes send a stream of preamble bytes (or wake up tones) equal to the polling period of their destination nodes, in order to ensure that they wake up in time to receive any actual data. In lightly loaded networks, these types of protocols can achieve better power savings than scheduling based protocols since the overhead associated with long contention periods and synchronization is avoided. In heavily loaded networks, however, the overhead of transmitting a long stream of preamble bytes starts to outweigh these benefits.

Hybrid protocols (SCP [Ye et al. 2006], Funneling-MAC [Ahn et al. 2006], and IEEE 802.15.4 [IEEE 2003]) combine some of the features present in TDMA, scheduled contention, and channel polling based protocols in order to find a reasonable tradeoff between message latency and power consumption. SCP, for example, combines the advantages provided by channel polling with those of scheduled contention in order to avoid the problems normally associated with requiring time synchronization while at the same time avoiding long preamble costs. Funneling-MAC, on the other hand, allows some nodes near a sink to run TDMA schedules while all others follow either a scheduled contention or polling based duty cycle. The overhead of

maintaining the TDMA schedule is mitigated by the fact that only a small number of nodes actually need to follow it.

To improve the performance of duty cycling protocols, *backbone* based power management protocols have also been proposed to support performance sensitive applications. Protocols such as ASCENT [Cerpa and Estrin. 2002], SPAN [Chen et al. 2001], GAF [Xu et al. 2001], and PEAS [Ye et al. 2003] set up a backbone of nodes that must be continually active in order to quickly forward messages between a source and its sink. Nodes not in the backbone are allowed to follow one of the four different types of duty cycling protocols presented in this section. Membership in the backbone is periodically changed in order to balance the power consumed by all nodes in the network.

In this paper we present a set of interfaces that allow the duty cycling protocols described above to be implemented in a platform independent manner at the link layer. In [Klues et al. 2007] we build on this work by creating an optimized, component based architecture within which these protocols can easily be implemented. The component-based architecture presented in [Klues et al. 2007] sits on top of the interfaces presented in this paper and is therefore complementary to them.

3. DESIGN OF THE ARCHITECTURE

In this section, we describe the architectural support necessary for providing flexible radio power management at the data link layer. This architecture defines a set of interfaces that allow different duty cycling protocols to be independently implemented on top of a common underlying radio stack. The introduction of these interfaces enables support for flexibly integrating different duty cycling protocols into a system based on its specific application requirements. This architecture also includes support for defining components that are capable of coordinating the duty cycling requirements from multiple applications. By combining these requirements inside a separate architectural component, different duty cycling protocols can be made to work with a diverse set of applications and platforms.

3.1 Overview of TinyOS

Although the architectural design presented in this section is not inherently tied to any particular OS, we use terminology and naming conventions borrowed from the TinyOS [Hill et al. 2000] operating system in order to describe it. TinyOS is the de facto standard operating system in use today for wireless sensor network development. Its component-based design offers a good, well-understood vocabulary for discussing interactions among the components in our architecture. For analogous reasons, the interfaces we present are all given in nesC [Gay et al. 2003] syntax, the implementation language of TinyOS.

Unlike traditional operating systems used in other types of distributed systems, TinyOS has no threads and is completely non-blocking. In a traditional operating system that support threads and blocking I/O, calls to long-running I/O operations do not return immediately and *block* an application from continuing execution until the operation is complete. In contrast, TinyOS implements all of its long running I/O operations using asynchronous I/O. It introduces the notion of a *split-phase* operation, separating the invocation and completion of an I/O operation into two separate phases of execution. An application begins an I/O operation by making

a function call directly into an I/O device driver. After the driver forwards this request to the I/O hardware it returns control immediately back to the application. The application is now free to continue executing or go to sleep while it waits for the I/O operation to complete. Once the I/O operation is complete, an interrupt is generated, triggering the device driver to issue a callback to the application so it can continue execution. In TinyOS terminology, the function that the application calls to begin the I/O operation is known as a *command*, and the callback is known as an *event*. All of the interfaces specified in this section are split-phase and thus are annotated with these terms.

Split-phase operations allow highly concurrent TinyOS applications to be implemented efficiently with a small memory footprint. They also enable fine-grained energy management to be implemented directly within the OS. Details on TinyOS and the advantages and disadvantages of using split-phase operations can be found in [Gay et al. 2007] and [Klues et al. 2007].

3.2 Supporting Flexibility

We begin our discussion on how our architecture is able to support flexibility by first defining a clear separation between what functionality is shared among duty cycling protocols and what is not. We define a set of common interfaces between the features present in radio specific MAC implementations and the features distinguishing one duty cycling protocol from another. Figure 1 shows how existing monolithic implementations are constructed, while Figure 2 shows how we separate them by introducing a set of uniform interfaces between a duty cycling protocol and its underlying MAC layer.

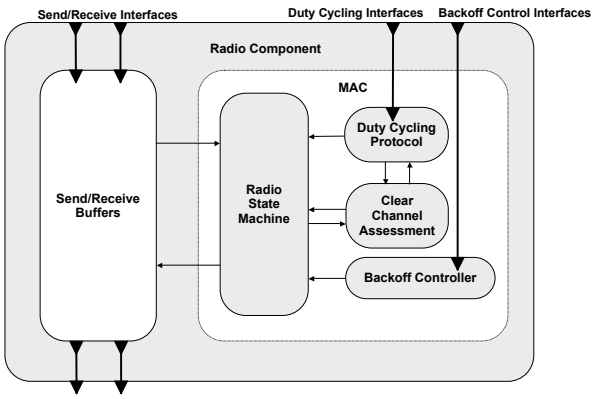

Fig. 1. Existing monolithic implementations of the MAC layer

In both figures we see 5 functional components that make up the MAC layer of a radio device driver: the radio state machine, a set of radio send/receive buffers, a clear channel assessment module, a backoff controller, and a duty cycling protocol. Each component is in charge of providing certain functionality required by the device driver, with the radio state machine overseeing all operations performed by each of the other components. When an application wishes to transmit a packet,

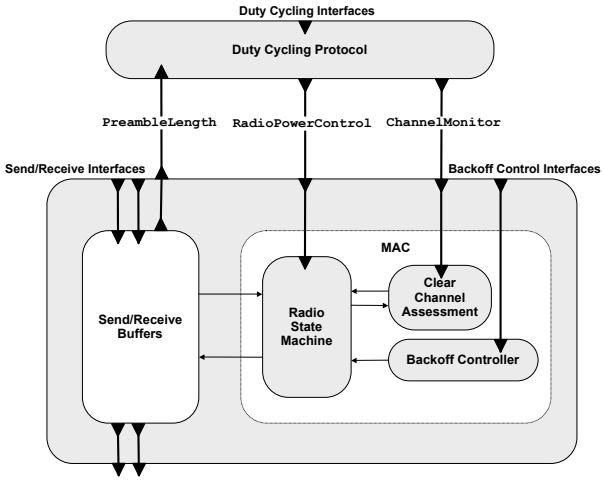

Fig. 2. Proposed separation of duty cycling protocols from common MAC level functionality

it is first inserted into a send buffer and only sent out over the radio once the radio state machine has decided it should be released. The radio state machine consults the backoff controller, clear channel assessment module, and duty cycling protocol in order to make this decision. The clear channel assessment module checks to make sure the radio channel is free for transmission, while the backoff controller decides how long a packet should be kept in the buffer before being sent over the radio (both initially and in the face of congestion on the radio channel). Finally, the duty cycling protocol is consulted to see if it imposes any restrictions on the packet being sent. Since the duty cycling protocol runs in parallel with all other radio driver functionality – transitioning the radio between its receive and idle power states – it may decide that a packet should be backed off a little while longer depending on its implementation.

Despite the modularized appearance of these components in the figure, most existing radio device drivers intertwine all of this functionality into a single unmanagable component. In particular, the implementation of the duty cycling protocol is tightly integrated with the basic MAC functions that could be reused by different duty cycling protocols. As a result, it is extremely difficult to maintain and upgrade existing duty cycling protocols as the underlying MAC implementation evolves and is ported to different platforms. Moreover, implementing new duty cycling protocols requires considerable effort on the part of the developer. They typically have to re-implement the entire radio driver from scratch, prohibiting improvements in MAC and power management technology from evolving independently. For example, in TinyOS, the default radio drivers for the Telosb and Mica2 mote platforms each have their own implementations of BMAC and its built in Low Power Listening capabilities. A single monolithic implementation exists for each platform, and common functionality is not shared between them.

In our proposed architecture, we explicitly remove the duty cycling protocol from basic MAC functionality and define a set of interfaces that allow them to interact with one another. Each interface is designed to have a *single* endpoint in some

higher level component, forcing any cross-layer protocols that use each interface to virtualize their functionality on top of them[3].Ideally, all MAC implementations will provide each interface so that they are not limited in the types of duty cycling protocols that can be implemented on top of them. Each interface is described in greater detail below.

The `RadioPowerControl` Interface:
The first of these interfaces is `RadioPowerControl`. This interface allows a radio to be switched between its *active* and *sleep* power states. It must be implemented by all types of MAC protocols, since without it, duty cycling of the radio is not possible.

```
interface RadioPowerControl {
  async command void on();
  async event void onDone( error );
  async command void off();
  async event void offDone( error );
}
```

The `ChannelMonitor` Interface:
The second interface is the `ChannelMonitor` interface. This interface is used to expose clear channel assessment (CCA) capabilities of a radio. This capability is required by all duty cycling protocols based on channel polling in order to determine if a radio channel has any activity on it or not. If this interface is not exposed, the use of certain duty cycling protocols (such as Low Power Listening [Polastre et al. 2004]) will not be possible.

```
interface ChannelMonitor {
  command void check();
  async event void free();
  async event void busy();
  event void error();
}
```

The `PreambleLength` Interface:
The third and final interface is the `PreambleLength` interface. This interface allows a duty cycling protocol to dynamically change the length of the preamble associated with a particular outgoing packet. It is intended for duty cycling protocols that require a constant wakeup tone of a certain length to be sent (transparent to the operation of the duty cycling protocol itself). For certain types of radios (in particular, packet radios like the cc2420 [Chipcon 2004]), the implementation of this interface may need to be emulated by sending multiple packets with very short gaps instead of a constant stream of preamble bytes. For duty cycling protocols that actually rely on the transmission of short packets, this interface is not necessarily required.

[3]Methods for supporting coordination of such cross layer protocols can be found in [Klues et al. 2007]

```
interface PreambleLength {
  async command void set( numBytes );
  async command uint16_t get();
}
```

In addition to the interfaces presented above, the MAC layer may also need to support time stamping for incoming and outgoing packets in order to facilitate the development of time synchronization protocols required by both scheduled contention and TDMA based protocols. Time stamp data can be appended as metadata to any packets passed up the radio stack through the `Receive` interface of the MAC layer. Duty cycling protocols that exploit the use of this information simply need to extract it from the packets as they are passed up the stack.

3.3 Supporting Multiple Applications

By exposing the interfaces described in the previous section, a multitude of different duty cycling protocols can be built on top of a common underlying radio stack.

With these duty cycling protocols in place, our architecture is now able to provide a framework for coordinating different power management requirements from multiple applications. For purposes of this discussion, an *application* refers to any component in the system that wishes to specify a set of power management requirements to any duty cycling protocol existing at the link layer. Our framework provides a mediator that resolves any conflicts among these applications' stated power requirements.[4] Figure 3 illustrates this framework for coordination.

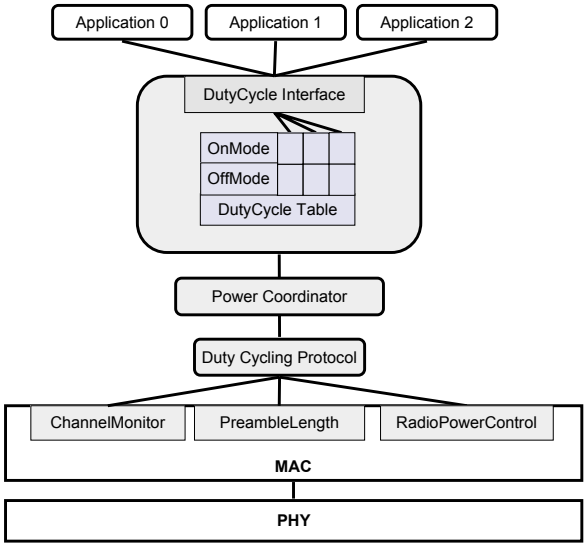

Fig. 3. Architecture for enabling the coordination of differing application requirements.

[4]If only a single application happens to be running on the system, this mediator need not be included in its compilation.

Applications insert parameters into a *Power Management Table* using a set of predefined interfaces. The interfaces depend on the power management protocol in use. Protocols based on scheduled contention provide interfaces that allow applications to specify their on/off intervals, while protocols based on channel polling provide interfaces that allow them to specify a polling period and a preamble length. Protocols based on a hybrid of these two approaches require both sets of interfaces. Rows in the Power Management Table represent a single parameter type into which an application may supply a value. Columns are used to separate the values supplied by different applications. For example, all parameters supplied by 'Application 1' in the figure will be stored in column 1, and all parameters supplied by 'Application 2' will be stored in column 2.

A *Power Coordinator* is used to coordinate the use of all parameters supplied to the Power Management Table by each application. It decides how to combine these parameters in order to provide a coherent radio power management solution that satisfies the needs of all applications as best it can. The Power Coordinator can be customized based on the requirements of the applications and the underlying duty cycling protocol on which they rely. We present two coordination policies in Section 5.2 as concrete examples that have been implemented in TinyOS.We envision that a library of common coordination techniques (similar to software design patterns) will be created that will be able to meet the needs of different types of applications.

4. IMPLEMENTATION

To enable the flexible integration of different radio power management protocols, the cc1000 and cc2420 radio stacks in TinyOS-2.x have been altered to expose the interfaces described in section 3.2. We have chosen to use TinyOS-2.0 as our implementation platform since it is still maturing and does not yet have many radio power management protocols developed for it. Our hope is that as developers start porting implementations of their protocols from TinyOS-1.x to TinyOS-2.x, they will do so within the architecture presented here. Doing so will ease the development of these protocols and create a rich set components for use within the architecture.

In this section, we describe the implementations of several different duty cycling protocols using the interfaces defined in section 3.2, as well as several coordination policies in our architecture. We note that the implementations provided in this section are just examples that demonstrate the flexibility and efficacy of the architecture presented in this paper. More sophisticated duty cycling protocols and coordination policies can be implemented within the framework provided here. policies, but rather the architectural framework within which they can be developed.

4.1 Duty Cycling Protocols

In this section we describe sample implementations of both the polling based Low Power Listening (LPL) protocol [Polastre et al. 2004] and a simple scheduling based protocol called Simple Synchronous Sleeping (SSS). A third protocol that we call Basic Synchronous Sleeping (BSS) is also introduced. It is functionally similar to SSS, but differs in the type of interface it provides to its users. By providing implementations of both polling based and scheduling based duty cycling protocols,

we are able to demonstrate the flexibility introduced by this new architecture.

4.1.1 *Low Power Listening.* LPL allows a radio to sleep for long periods of time, periodically polling the radio channel to check if there are any incoming packets. If no packet is present, it goes back to sleep for an amount of time equal to the node's polling interval. Packets are sent with preamble lengths equal to the size of the polling interval so that the destination node is guaranteed to be awake when the packet is sent. LPL allows a user to specify two different parameters: the time interval between subsequent checks for activity on the radio channel, and the preamble length for outgoing packets. We have implemented Low Power Listening on top of the interfaces presented in section 3.2. Figure 4 shows our new implementation of LPL as a separate component, and how it uses the standard interfaces defined in the previous section to interact with the modified B-MAC that no longer includes LPL.

This figure depicts three situations in which the new LPL may use the interfaces provided by the MAC layer to perform its sleep scheduling duties. During startup (Figure 4(a)), an application specifies the LPL mode of operation (i.e. the check interval to use along with its corresponding preamble length) through the `Lpl` interface provided by LPL (1). The MAC layer then retrieves the newly specified preamble length through the `PreambeLength` interface (2), and LPL turns the radio off through the `RadioPowerControl` interface (3). Once the radio has been completely shut down (4), a timer is set based on the check interval specified by the application (SLEEP_TIME)(5).

One of two conditions could then occur. In both cases (Figure 4(b), 4(c)) the timer will expire (1) and the channel will be checked for activity (2). An event will then come back signifying that the channel is either busy or free (3), and the timer will be reset with one of two values. If the channel was free (Figure 4(b)), the timer is reset to its check interval length (SLEEP_TIME)(4) and the radio is shut off (5)(6). If the channel is busy, however,(Figure 4(c)) the timer is set to allow an entire packet to be received (MAX_PACKET_LENGTH)(4), and the radio is turned fully on (5)(6).

4.1.2 *Scheduling based Duty Cycling Protocols.* We have also designed and implemented a scheduling based duty cycling protocol known as SSS (Simple Synchronous Sleeping) on top of our interfaces. SSS relies on global time synchronization of all nodes in a network to precisely control their duty cycles. This protocol is intended as an example of how a scheduling based protocol *could* be implemented, rather than a replacement for more robust protocols. This protocol allows the duty cycle of the radio to be tuned through the following interface:

```
interface RadioDutyCycling {
  command error_t setDutyCycle( on, off );
  command error_t setOnTime( onTime );
  command error_t setOffTime( offTime );
  event void beginOnTime();
  event void beginOffTime();
}
```

A higher layer uses this interface to set the duty cycle of the radio and be notified whenever it has been switched on or off. Since the start of every radio's duty

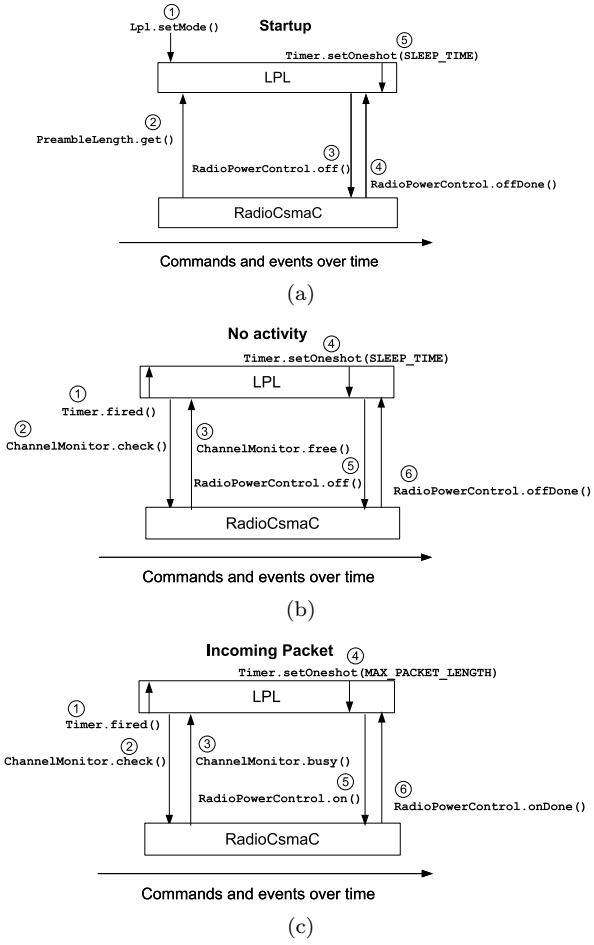

Fig. 4. Interaction of platform independent LPL implementation with radio level interfaces. The `RadioCsmaC` component represents a MAC layer with CCA capabilites built into it, and therefore provides the `ChannelMonitor` interface.

cycle must be synchronized, all nodes having the same duty cycle will be able to communicate with each other during the on time of the radio (using CSMA/CA) and conserve energy during the off time. Figure 5 shows our implementation of SSS and how it uses the interfaces presented in the previous section to interact with the radio.

During startup (Figure 5(a)), an application specifies the mode of operation for SSS (length of on and off times within a single duty cycle period) through the `RadioDutyCycling` interface (1). A timer is then set to the on time specified by the application (2), and the radio is switched on (3). Once the radio has been fully switched on (4), the application is signaled notifying it of this event (5). After this timer expires, SSS alternates the on and off states of the radio according to the time intervals specified by the application. The steps taken in each situation can

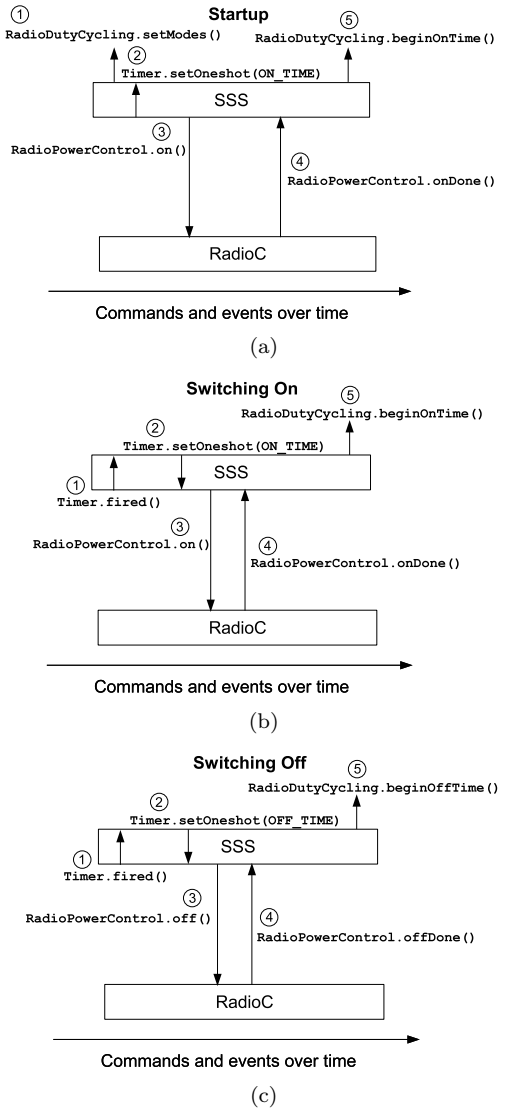

Fig. 5. Interaction of platform independent SSS implementation with radio level interfaces. The `RadioC` component represents a MAC layer which doesn't necessarily provide CCA capabilites, and therefore doesn't need to provide the `ChannelMonitor` interface.

be seen in in Figure 5(b) and Figure 5(c) respectively.

The third protocol we have implemented is BSS (Basic Synchronous Sleeping). Both SSS and BSS require time synchronization for all nodes in a network, and they both turn the radio on and off for certain time durations as specified by the user. The primary difference between the two is that SSS allows an application to specify a periodic radio duty cycle, while BSS requires that an application explicitly request the radio to be turned on or off just before making each transition. This allows the

on and off periods to be changed on every transition rather than requiring them to be periodic. SSS could, in fact, be simulated using BSS by always specifying the same on and off times every time the appropriate transition was supposed to occur. Because of this flexibility in changing on and off times at every transition, BSS has the capability of supporting more sophisticated scheduling algorithms as well as integrating different schedules supplied by multiple applications.

An application can inform BSS when (as well as for how long) to power the radio on and off using the following interface:

```
interface DutyCycleTimes {
  command turnOnFor( onTime );
  command turnOffFor( offTime );
  event void ready();
}
```

Calling the `turnOnFor()` and `turnOffFor()` commands does not necessarily indicate that the radio will be turned on or off immediately. These commands are a way of specifying the on and off times that will be used the next time BSS signals the `ready` event. To duty cycle a radio, an application can alternate calls to `turnOnFor()` and `turnOffFor()` within the body of the `ready()` event. If no calls to `turnOnFor()` or `turnOffFor()` are made between subsequent `ready()` events, the power state of the radio remains unchanged and the next `ready()` event is signaled after the same amount of time it took to receive the previous `ready()` event.

Our current implementations of SSS and BSS use a simple time synchronization scheme that operates in a single-hop network. Under this scheme, all nodes other than the base station turn on their radio interfaces once they are booted, and leave it on until they have a received a synchronization packet. Once the base station is started, it sends the synchronization packet, and all nodes in the network start a timer simultaneously. There are no synchronization updates and clock drift is not taken into consideration. If one of the nodes happens to not hear the synchronization message, then it will not become synchronized and the entire network will need to be rebooted. It is left as future work to implement more sophisticated synchronization protocols using the time stamping features provided by the MAC layer [Elson et al. 2002; Ganeriwal et al. 2003; Maroti et al. 2004].

4.1.3 *Discussion.* In addition to the simple duty cycling protocols presented here, more complicated duty cycling protocols such as TRAMA, DRAND, 802.15.4, and SCP-MAC can also be implemented within this architecture. We consider these protocols as examples because DRAND is a pure TDMA based protocol, TRAMA is a TDMA based protocol that contains some elements of scheduled contention, and both 802.15.4 and SCP-MAC are representative hybrid protocols that combine features present in TDMA, scheduled contention, and polling based protocols. In addition to the interfaces provided by our architecture, TRAMA, 802.15.4, and SCP-MAC would use the time stamping features provided by the MAC order to implement their time-synchronization components. These three protocols would also use the underlying CSMA/CA capabilities of the radio to contend for use of the radio channel when appropriate.

TRAMA and DRAND only require the use of the `RadioPowerControl` interface to turn the radio on and off in the slots setup by the TDMA schedules they produce. In the case of TRAMA, these would be the *transmission slots* setup for scheduled transmission of packets between all nodes; packets sent in TRAMA's *signaling slots* would contend for the channel using the CSMA/CA capabilities of the radio. In the case of DRAND, collisions are allowed between nodes during the initial setup phase and all subsequent transmissions are made according to the TDMA schedule that results from it.

802.15.4 uses the `RadioPowerControl` interface to turn on the radio for both its contention access period (CAP) and its contention free period (CFP). During the CAP it would use CSMA/CA to send and receive messages over the radio and during the CFP it would use a TDMA schedule to decide in which guaranteed times slots a node should be sending. After both the CAP and CFP, the radio would be turned off again via the `RadioPowerControl` interface.

SCP would only use the `RadioPowerControl` interface to turn the radio on for the reception of packets, and off again once all packets were received. It would additionally require the `ChannelMonitor` interface to perform its channel polling capabilities and the `PreambleLength` interface to adjust the number of preamble bytes associated with any outgoing packets. More details on a full implementation of SCP, as well as four other CSMA and TDMA based protocols built on top of the architecture presented here can be found in [Klues et al. 2007].

4.2 Coordination Policies

We now present how two different coordination polices can be implemented using the mechanisms described in section 3.3. We note that different systems may desire different coordination policies. We provide the two coordination policies just as examples to demonstrate the capability of our architectural framework to coordinate multiple power management requirements. One policy coordinates the use of different duty cycling requirements from multiple applications, while the other coordinates a backbone maintenance protocol with these same duty cycling requirements. We note that these policies are only examples used to demonstrate the flexibility of our architecture in incorporating different coordination policies. In practice the choice of coordination policies is dependent on the specific application requirements and the duty cycle protocol.

4.2.1 *Combining Duty Cycle Requirements.* We have implemented a coordination policy that combines different duty cycling requirements specified through the `RadioDutyCyling` interface. The parameters passed to this interface are stored in a *Power Management Table*, and a *Power Coordinator* is used to combine these requirements to produce a single coherent duty cycling schedule. This schedule is used to inform BSS of the on and off times it should use when duty cycling the radio. In the implementation, the Power Coordinator aggregates the duty cycles specified by multiple applications according to an OR policy as shown in Figure 6.

This coordination policy is as follows. (1) All duty cycles from all applications are synchronized to begin at the same time instant. (2) They all run periodically according to their own schedule. (3) If *any* of the duty cycles requires the radio to be on at any particular point in time, the radio will be turned on. (4) Only if

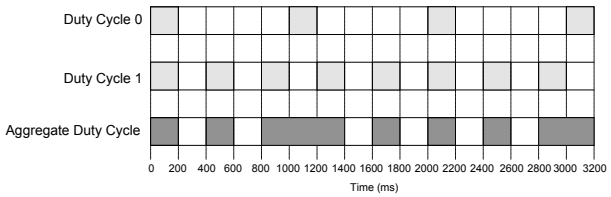

Fig. 6. Aggregation of multiple duty cycles

all duty cycles indicate that the radio should be turned off will the radio ever be turned off.

There exist two different approaches to implement the above coordination policy. First, the coordinator can pre-compute and store a periodic schedule whose length equals the least common multiple (LCM) of all applications. The on and off times of the radio can be obtained by efficient table lookups from the schedule. In the second approach, the coordinator can compute the on and off times of the radio at each transition point and inform BSS on the fly. Since no pre-computed schedule needs to be stored, this approach minimizes the storage overhead. However, it imposes high computational complexity when the number of different duty cycles is large. In this work, we adopt the first approach. Figure 6 illustrates the pre-computed schedule. We can see that 'Duty Cycle 0' has an on time duration of 200ms and an off time duration of 800ms, while 'Duty Cycle 1' has both an on and off time duration of 200ms. The period of 'Duty Cycle 0' is therefore 1000ms, while the period of 'Duty Cycle 1' is just 400ms. In order to find the period of the 'Aggregate Duty Cycle' schedule, the least common multiple of all duty cycles must be determined. In this case it is 2000ms. Since multiple on and off periods will exist within one period, BSS is more appropriate than SSS for executing the aggregated schedule.

4.2.2 *Coordination between Duty Cycling and Backbone Protocols.* The second coordination policy allows a backbone maintenance protocol to work together with duty cycling protocols. Backbone protocols save energy by controlling the spatial density of active nodes, while duty cycling saves energy by controlling the temporal frequency of radio activity. By combining the energy benefits provided by the two different types of protocols, this coordination policy is able to save more energy than using of either one of these protocols individually.

The backbone based protocol we use is known as PEAS [Ye et al. 2003]. We now briefly discuss the basic operation of PEAS and the details can be found in [Ye et al. 2003]. When nodes first wake up in a PEAS enabled network, they send out a probing message to determine if any of their neighboring nodes are awake and operating. If they do not hear any responses, they decide to become *active* and turn their radios on accordingly. Once a node has become active, it will remain active until its power supply has been depleted. Active nodes take on the responsibility of responding to probing messages sent by inactive nodes. If inactive nodes hear one of these responses, they return immediately to sleep and wait some predetermined amount of time before sending out the next probe. The amount of time they have to wait changes dynamically based on the number of active nodes within their probing

range as well as the frequency with which other inactive nodes send out their probing messages. As a result, the network maintains a backbone composed of active nodes while all other nodes run in a low duty cycle. We have implemented a lightweight version of PEAS that uses the same probe/reply mechanism as described above, but uses a fixed delay time between each probing message.

We now describe an application scenario in which PEAS can be combined with a duty cycling protocol to achieve significant more energy saving while still meeting the spatial and temporal sampling requirements of the application. Suppose a data collection application requires the temperature of a region to be reported to the base station every 10 seconds. This requirement can be translated into a radio duty cycle. For instance, in a period of 10 seconds, a data collecting node only wakes up for 2 seconds (to transmit its own data and forward the data from neighbors) and remains asleep for 8 seconds. This duty cycle is referred to as *application duty cycle* hereafter. In addition, the application specifies a minimum spatial granularity of the gathered data in the form of "one sample in every 10 m range". To reduce radio activity, PEAS can be run to maintain a set of active nodes whose probing range is 10 m while all the other nodes operate in a low duty cycle, e.g., wake up for one second in every 20 seconds. This duty cycle is referred to as *PEAS duty cycle* hereafter. To maximize the network energy saving, the backbone nodes chosen by PEAS can operate in the application duty cycle. As discussed previously, the backbone nodes must exchange messages with inactive nodes when they wake up and hence they must run the PEAS duty cycle as well. Therefore, the application and PEAS duty cycles need to be combined on the backbone nodes while only PEAS duty cycle is needed for inactive nodes. Fig. 7 illustrates a network with backbone and inactive nodes, as well as the duty cycles run by them.

The coordination between PEAS and the duty cycle protocol that is described above can be easily implemented in our architecture. Fig. 8 shows the instantiation of our power management framework on a backbone node when both protocols run simultaneously. The *power management table* has three columns that contain the parameters of three different duty cycles. Note that these parameters can be filled in through the *DutyCycle* interface separately by both PEAS and the sensing application themselves. As the table shown in the figure is for a backbone node, a special duty cycle, *PEAS active*, whose *OffTime* is zero is specified in the table. This indicates that the node should remain active if only PEAS is running. Note that PEAS will not specify the PEAS active duty cycle in the power management table if it determines that the node is not a backbone node. The *power coordinator* uses the following logic to combine PEAS with the duty cycle protocol. When a duty cycle with zero *OffTime* is found in the power management table, the coordinator aggregates the PEAS duty cycle and the application duty cycle according to the same OR policy as described in Section 4.2.1. Otherwise, the node is an inactive nodes and only PEAS duty cycle is used. The duty cycle output by the power coordinator is then passed to BSS that switches the radio on/off accordingly.

A key advantage of combining PEAS and the duty cycling protocol within the architecture is that neither the implementations of PEAS, BSS, nor the duty cycling protocol needs to be altered in order to achieve these energy savings. All coordination is handled by the implementation of the coordination policy itself.

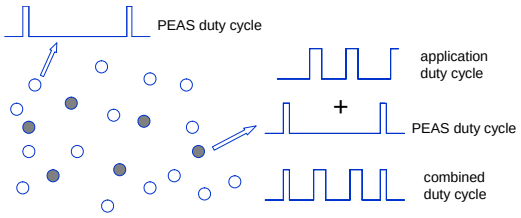

Fig. 7. A network that runs PEAS and a duty cycle protocol simultaneously. Grey nodes denote the backbone nodes that operate in a combined duty cycle. White nodes denote inactive nodes that only run PEAS duty cycle.

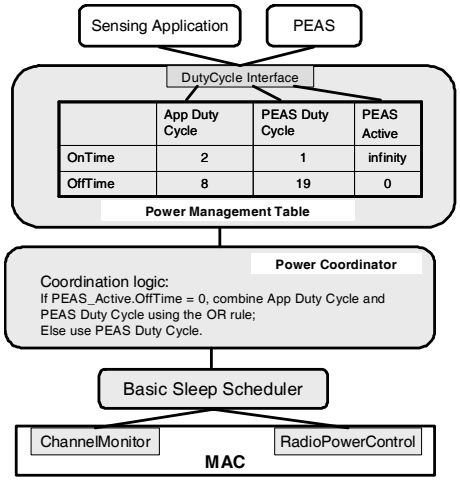

Fig. 8. Combining the PEAS protocol and the duty cycle requirement of a sensing application on a backbone node.

5. EVALUATION

This section presents the empirical evaluation of our architecture. The first set of experiments evaluates the efficiency of the uniform interfaces between duty cycling protocols and the MAC in terms of both performance and code size. The second set of experiments evaluates the effectiveness of our framework in coordinating the duty cycle requirements of multiple applications. The final set of experiments evaluates the coordination of a backbone protocol and duty cycle requirements of multiple applications.

5.1 Efficiency

The first set of experiments involve comparing the monolithic implementation of Low Power Listening (LPL) on the cc1000 radio stack with the one designed for use within our architecture. Our experimental settings are the same as the ones presented for the full B-MAC implementation in [Polastre et al. 2004]. We also compare the difference in the code size between the two implementations. By showing that our implementation of Low Power Listening is comparable to the original one in terms of both performance and code size, we are able to demonstrate

that our architecture provides just as efficient a framework for implementing it as the original one. Since our implementation is not implemented within the MAC component of the default radio stack, however, it demonstrates the level of flexibility made possible by our architecture.

We first measure throughput vs. number of nodes in a single hop network. There is one receiver, with a variable number of senders from 1 to 4. All senders are equidistant from the receiver at 2 feet. Each sender transmits as often as possible with messages containing 38 bytes of data and 8 preamble bytes. We measure the total throughput (bits per second) at the receiver over 2 minutes. The results are given in Figure 9.

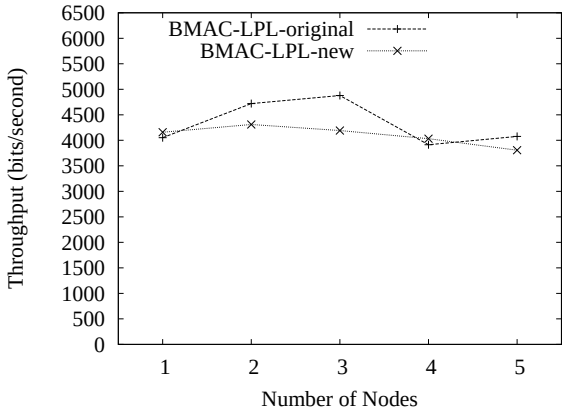

Fig. 9. Throughput vs. Number of nodes at 100% duty cycle for two different LPL implementations

With only one node in the network, performance between the two implementations is almost identical. While the original LPL implementation achieves slightly higher throughput for two and three nodes, the results for the two implementations are comparable again for four and five.

Next, we measure the delivery latency vs. number of hops in a fixed route multihop network. Nodes are placed in a chain, with the first node being both the source and the sink node. Messages are sent from one node to the next until the last node in the chain is reached. Messages are then sent in reverse back to the original sender. The number of nodes varies from 2 to 5, resulting in 2, 4, 6, and 8 hops respectively. The sender sends 20 messages, each containing 38 bytes of payload and a variable number of preamble bytes depending on the length of the LPL check interval that has been selected. LPL check intervals of "always on", 800ms, and 1600ms were chosen, and the average latency from source to sink of each data packet was measured. The results are given in Figure 10. In this experiment, we see that the results for both LPL implementations are identical for all measured check intervals.

Table I shows the difference in code size for each implementation when they are compiled into the applications used in the experiments above. As expected, both the

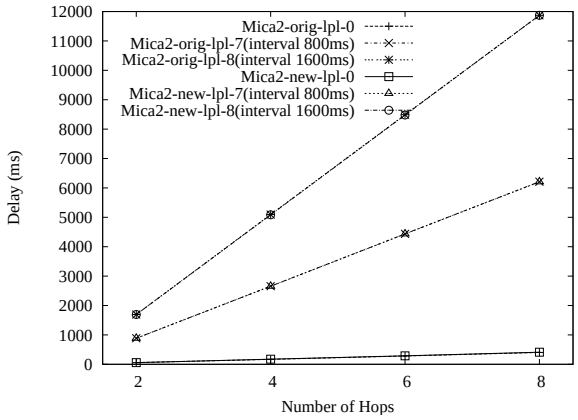

Fig. 10. Latency vs. Number of Hops at different check intervals for two different LPL implementations

	Original LPL	New LPL
	RAM/ROM	RAM/ROM
SenderApp	383/11956	394/12350
ReceiverApp	705/15098	716/15560

Table I. LPL Memory Footprint (Bytes)

RAM and ROM sizes for the new implementation are slightly larger than for original one. This extra overhead is less than 5% in both cases, however, and therefore negligible. Such a small increase in code size is possible due to the aggressive cross-component inlining and other optimizations made available by the full-program analysis of TinyOSs nesC compiler. The main contributor increase we actually see comes from the extra timer required by the new LPL implementation. In the original implementation of LPL, the timer used to switch between the different states of the radio was shared by the LPL implementation. Other contributors include additional flags and logic needed to coordinate the use of the interfaces now provided by the radio stack.

The second set of experiments shows the performance characteristics of our SSS implementation. The results of these experiments show that it is easy to reuse the implementation of this sleep scheduling policy on top of two very different radio stack implementations. Results are given for both Mica2 and TelosB.

The setups for each experiment are the same as those used for LPL. For measuring throughput vs. number of nodes, we ran SSS at duty cycles of 100%, 47%, and 20%, and measured the total throughput for each duty cycle over 2 minutes. For measuring latency vs. number of hops, we ran SSS at a 50% duty cycle, and measured the average latency from source to sink for a single packet. Each experiment was ran multiple times, and the numbers presented are an average of each of those runs.

Figures 11 and 12 show that SSS is able to deliver more data as its duty cycle is increased. As expected, TelosB achieves higher throughput for all duty cycles in

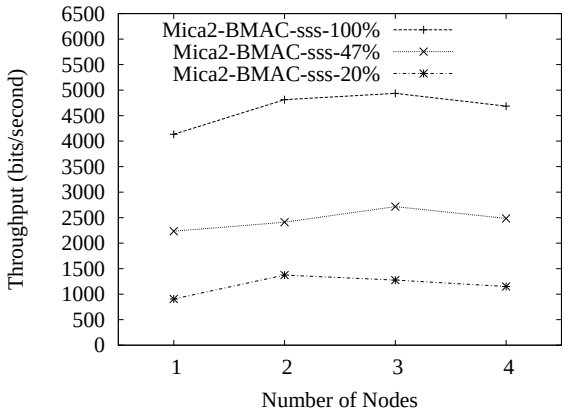

Fig. 11. Throughput vs. Number of nodes at different duty cycles for the SSS implementation on mica2

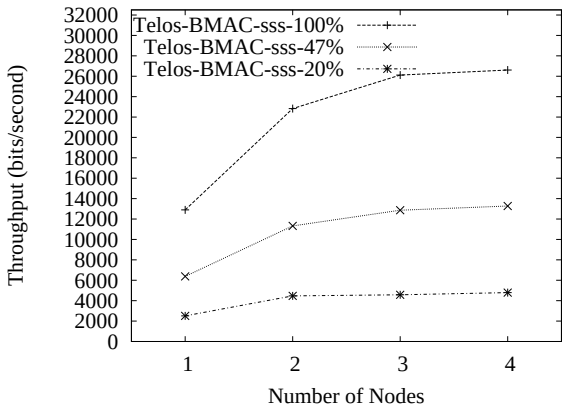

Fig. 12. Throughput vs. Number of nodes at different duty cycles for the SSS implementation on telosb

both experiments because data is sent at a much higher rate by the cc2420 radio than by the cc1000 radio.

Figure 13 shows the latency associated with running SSS. Once again, the higher data rate of the cc2420 radio accounts for the difference in performance between the TelosB and the Mica2 platforms in this experiment.

Overall, the results of these experiments show the following: (1) Implementing LPL using our framework incurs a negligible performance penalty. (2) Exposing the proposed MAC layer interfaces produces a slight increase in code size, but allows much more flexibility in choosing the most appropriate duty cycling protocol. (3) Both channel polling and scheduled contention protocols can easily be implemented on top of these interfaces and used by different mote platforms. These implementations produce results typical of these types of protocols.

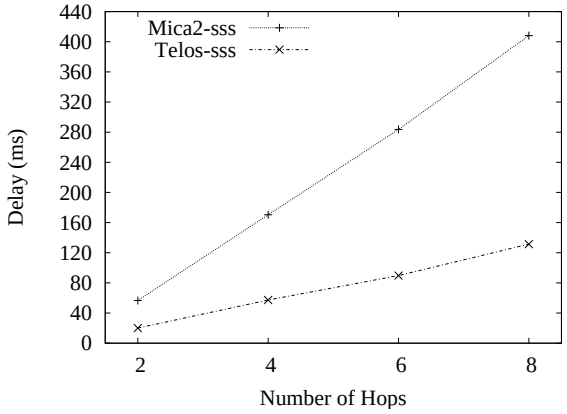

Fig. 13. Latency vs. Number of Hops at 50% duty cycle for the SSS implementation on both mica2 and telosb

5.2 Coordinating Multiple Duty Cycles

In this section, we evaluate the architecture in terms of the different coordination policies used to combine duty cycles specified by multiple applications. The first experiment demonstrates the ability to combine the duty cycling requirements of multiple applications in a way that is transparent to each of them. The second one presents the results obtained from coordinating the duty cycling requirement from multiple applications for use with PEAS and BSS. The network used in this set of experiments is a one-hop cluster consisting of a master TelosB node and a number of slave TelosB nodes. Each slave node runs an application that periodically sends packets to the master node. Each application under test is simply a dummy application that sets a duty cycle on the radio, takes a sensor reading, and flushes bytes out over the radio at specific intervals.

Although each node only runs a single application, up to 6 different applications can be running in the network at any given time. The on time of the duty cycle for each application is 200ms, with off times of 200ms, 600ms, 1.4s, 3s, 6s, and 12.6s, respectively. Each application sends a packet of 66 bytes (including header and payload) at a random time within the 200ms active period of each duty cycle. The master node is able to receive packets from each application by running an aggregate duty cycle according to the OR policy described in section 3.3.

The first run of experiments consists of the master node and two slave nodes running the application with the lowest duty cycle. Two more slave nodes running the application with the next highest duty cycle are then added in each following run. Each run lasts for 320 s and is performed only once. Figure 14 shows the delivery ratio measured at the master node for each run. We can see that the delivery ratio remains close to 100% as the number of applications increases. Once all six applications (total 12 nodes) have been added to the network, however, we do begin to see a slight increase in the number of packets that are lost.

Figure 15 shows the duty cycle measured at the master node. A 100% duty cycle corresponds to the radio always being on, and a 0% duty cycles corresponds to the

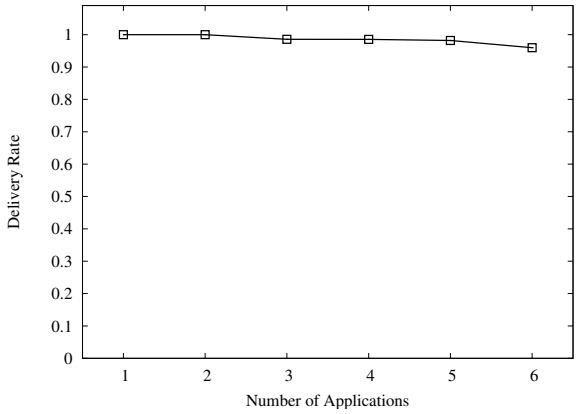

Fig. 14. The delivery ratio measured at the master node.

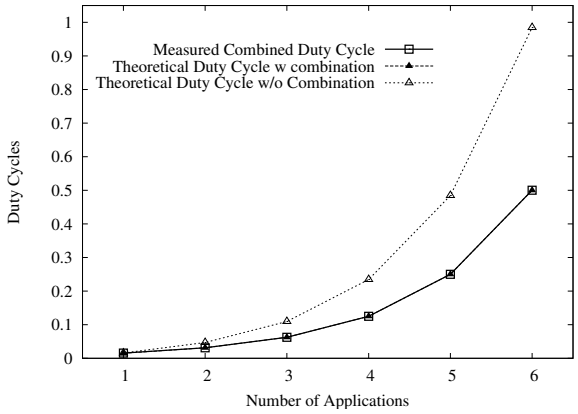

Fig. 15. The duty cycle of the master node.

radio always being off. The duty cycle was calculated by instrumenting the cc2420 radio stack with a 32 KHz timer in order to measure the amount of time spent in each radio state. We can see that the duty cycle measured at the master node matches the predicted curve, verifying the correctness of the combination logic of the aggregator. As a baseline we also show the predicted duty cycle of the master node if no aggregation policy were used. This duty cycle is simply calculated as the sum of the duty cycles of all applications in the network. As shown in Figure 15, performing the combination will always yield a lower overall duty cycle than not performing it.

The results in this section demonstrate that this coordination policy is capable of correctly combining the duty cycles specified by multiple applications, and that combining these duty cycles according to some aggregation policy can potentially lead to lower energy consumption.

5.3 Coordinating Duty Cycles with PEAS

In this section, we evaluate the architecture in terms of the coordination policy allowing PEAS to be combined with applications specifying different duty cycles. The network consists of a master Telosb node and 15 slave Telosb nodes placed within a 5×3 grid. Each slave node runs both PEAS as well as one of six different applications. Each application runs with a duty cycle period of 3.2s. The on times of each duty cycle range from 200ms to 1.2s in steps of 200ms. Nodes designated as 'inactive' by PEAS run with a duty cycle period of 16s and an on time of 200ms. Inactive nodes send probe messages at some random time within their on periods, and active nodes send messages from their applications at some time during their on periods. Although all nodes are within communication range of one another, the probing range of PEAS is limited to 1.5 times the grid width.

In this set of experiments we measure the total energy consumed by the radio for all nodes in the network. The amount of energy used by each radio is measured as the sum of the energy consumed in each of four different radio states: idle, receiving, transmitting, and sleeping. We first measure the total time that the radio spends in each radio state by instrumenting the Telosb cc2420 radio stack with a 32 KHz timer. We then calculate the energy consumed in each state by multiplying the total time the radio spends in that state by the power consumed in that state. These power consumption values are all taken directly from the cc2420 data sheet [Chipcon 2004][5].

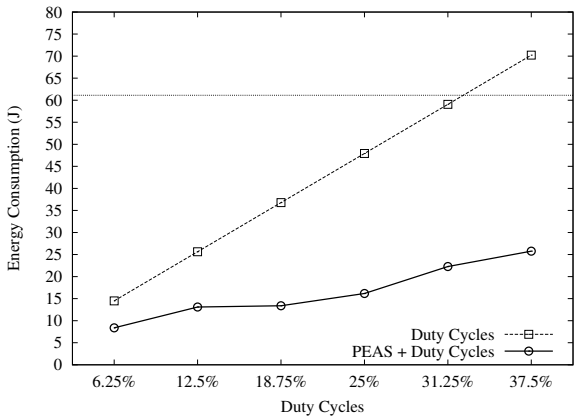

Fig. 16. The total energy consumption of the network.

As a baseline, we first measure the power consumption of the network when only PEAS is enabled and all active nodes have their radios powered all the time. We then disable PEAS and allow all nodes to run their duty cycling applications, without PEAS specifying certain nodes as inactive. Finally, we measure the power

[5]There are two different sleeping modes available on the cc2420. In the sleeping mode benchmarked here, the transmitter is turned off while the crystal oscillator and voltage regulator remain on. In the data sheet this state is referred to as IDLE.

consumption when these two policies are combined: PEAS is enabled and only active nodes are able to run their application. All experiments were ran multiple times, and the numbers presented are an average of these runs. Figure 16 shows the total energy consumption of the network in each situation. The baseline is represented by a straight line.

We can see from Figure 16 that the policy combining PEAS with each application duty cycle yields the lowest energy consumption. These energy savings are achieved by (1) allowing PEAS to choose the subset of nodes that will actually run each application, and (2) allowing those nodes chosen by PEAS to run at their application-specified duty cycle. Overall, the energy consumption under the combined policy is $57 - 86\%$ lower than running PEAS alone and $42 - 63\%$ lower than duty cycling alone.

The results in this section demonstrate the power of combining complementary power management protocols using our architectural framework. By using the Power Coordinator to combine the use of the these protocols, more energy can be saved than by using either one of them individually.

6. CONCLUSION

In this paper, we have presented link layer support towards realizing a unified radio power management architecture for use in wireless sensor networks. The architectural support we have presented is comprised of two key components: (1) a set of standard interfaces allowing different duty cycling protocols to be implemented on top of a common radio stack, and (2) an abstraction layer that can incorporate the use of different coordination polices for coordinating the duty cycling requirements of multiple applications running on the system.

We have demonstrated the flexibility of this architecture through the development of several different duty cycling protocols, and have conducted experiments in order to evaluate their performance. We have also shown how the architecture can be used to coordinate the duty cycling requirements of multiple applications. By replacing only the coordination policy, we demonstrate that it is possible to coordinate these requirements with those of a backbone based power management protocol as well.

In the future, we plan to build on the ideas presented in this paper to support power management protocols existing at layers other than the link layer. Specifically, we plan to add support for transmission power control protocols existing at the network layer. Another important direction for future work is to integrate this architecture with an overall sensor network architecture [Berkeley][USC]. Additionally, we plan to integrate this architecture with the power management techniques used by other hardware components (e.g., microcontrollers, sensors, and flash) on a wireless sensor network platform [Klues et al. 2007].

7. ACKNOWLEDGEMENTS

This work is supported by NSF NeTS-NOSS grant #CNS-0627126.

REFERENCES

Ahn, G.-S., Miluzzo, E., Campbell, A. T., Hong, S. G., and Cuomo, F. 2006. Funneling-mac: A localized, sink-oriented mac for boosting fidelity in sensor networks. In *Sensys*.

Berkeley, U. a network architecture for wireless sensor networks. http://webs.cs.berkeley.edu/SNA/.

Buettner, M., Yee, G. V., Anderson, E., and Han, R. 2006. X-mac: A short preamble mac protocol for duty-cycled wireless sensor networks. In *Sensys*.

Cerpa, A. and Estrin., D. 2002. Ascent: Adaptive self-configuring sensor networks topologies. In *INFOCOM*.

Chen, B., Jamieson, K., Balakrishnan, H., and Morris, R. 2001. Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks. In *MobiCom*.

Chipcon. 2004. Cc2420 radio data sheet.

El-Hoiyi, A., Decotignie, J.-D., and Hernandez, J. 2004. Wisemac: an ultra low power mac protocol for the downlink of infrastructure wireless sensor networks. *Computer Communications 1*, 244– 251.

Elson, J., Girod, L., and Estrin, D. 2002. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. ACM Press, New York, NY, USA, 147–163.

Ganeriwal, S., Kumar, R., and Srivastava, M. B. 2003. Timing-sync protocol for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*. ACM Press, New York, NY, USA, 138–149.

Gay, D., Levis, P., and Culler, D. 2007. Software design patterns for tinyos. *ACM Transactions on Embedded Computing Systems (TECS) 6,* 4.

Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D. 2003. The nesc language: A holistic approach to network embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*.

He, T., Krishnamurthy, S., Luo, L., Yan, T., Gu, L., Stoleru, R., Zhou, G., Cao, Q., Vicaire, P., Stankovic, J. A., Abdelzaher, T. F., Hui, J., and Krogh, B. 2006. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw. 2,* 1, 1–38.

Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D. E., and Pister, K. S. J. 2000. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*. 93–104.

IEEE. 2003. Wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal area networks (lr-wpans). In *IEEE Standard 15.4*.

Klues, K., Hackmann, G., Chipara, O., and Lu, C. 2007. A component based architecture for power-efficient media access control in wireless sensor networks. In *In proceedings of the Fifth ACM Conference on Embedded Network Systems (Sensys'07)*. Sydney, Australia.

Klues, K., Handziski, V., Lu, C., Wolisz, A., Culler, D., Gay, D., and Levis, P. 2007. Integrating concurrency control and energy management in device drivers. In *In proceedings for The 21st ACM Symposium on Operating Systems Principles (SOSP 2007)*. Stevenson, WA.

Klues, K., Xing, G., and Lu, C. 2007. Towards a unified radio power management architecture for wireless sensor networks. In *In proceedings of the First International Workshop on Wireless Sensor Network Architecture (WWSNA'07)*. Cambridge, MA.

Maroti, M., Kusy, B., Simon, G., and Ledeczi, A. 2004. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM Press, New York, NY, USA, 39–49.

Polastre, J., Hill, J., and Culler, D. 2004. Versatile low power media access for wireless sensor networks. In *SenSys*.

Rajendran, V., Obraczka, K., and Garcia-Luna-Aceves, J. J. 2003. Energy-efficient collision-free medium access control for wireless sensor networks. In *SenSys*.

Rhee, I., Warrier, A., Aia, M., and Min, J. 2005. Z-mac: a hybrid mac for wireless sensor networks. In *SenSys*.

Rhee, I., Warrier, A., Min, J., and Xu, L. 2006. Drand: distributed randomized tdma scheduling for wireless ad-hoc networks. In *MobiHoc*.

Santi, P. 2005. Topology control in wireless ad hoc and sensor networks. *ACM Comput. Surv. 37,* 2.

Szewczyk, R., Mainwaring, A., Polastre, J., Anderson, J., and Culler, D. 2004. An analysis of a large scale habitat monitoring application. In *SenSys*.

Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., Burgess, S., Dawson, T., Buonadonna, P., Gay, D., and Hong, W. 2005. A macroscope in the redwoods. In *SenSys*.

USC. A architecture for tiered wireless sensor networks. http://enl.usc.edu/projects/tenet/.

van Dam, T. and Langendoen, K. 2003. An adaptive energy-efficient mac protocol for wireless sensor networks. In *SenSys*.

Xu, Y., Heidemann, J., and Estrin, D. 2001. Geography-informed energy conservation for ad hoc routing. In *MobiCom*.

Ye, F., Zhong, G., Lu, S., and Zhang, L. 2003. Peas: A robust energy conserving protocol for long-lived sensor networks. In *ICDCS*.

Ye, W., Heidemann, J., and Estrin, D. 2004. Medium access control with coordinated, adaptive sleeping for wireless sensor networks. *IEEE/ACM Transactions on Networking*.

Ye, W., Silva, F., and Heidemann, J. 2006. Ultra-low duty cycle mac with scheduled channel polling. In *SenSys*.