

A Stream-Oriented Power Management Protocol for Low Duty Cycle Sensor Network Applications

Nithya Ramanathan[†][◇], Mark Yarvis[◇], Jasmeet Chhabra[◇], Nandakishore Kushalnagar[◇],
Lakshman Krishnamurthy[◇], Deborah Estrin[†]

[◇] Intel Corporation
2111 N.E. 25th Ave, Portland, OR
{mark.d.yarvis, jasmeet.chhabra,
nandakishore.kushalnagar,
lakshman.krishnamurthy}@intel.com

[†] UCLA, Center for Embedded Network Sensing
3563 Boelter Hall
Los Angeles, CA
{nithya, destrin}@cs.ucla.edu

Abstract

Most power management protocols are packet-based and optimize for applications with mostly asynchronous (i.e. unexpected) traffic. We present AppSleep, a stream-oriented power management protocol for latency tolerant sensor network applications. For this class of applications, AppSleep demonstrates an over 3x lifetime gain over B-MAC and SMAC. AppSleep leverages application characteristics in order to take advantage of periods of high latency tolerance to put the network to sleep for extended periods of time, while still facilitating low latency responses when required. AppSleep also gives applications the flexibility to efficiently and effectively trade latency for energy when desired, and enables energy efficient multi-fragment unicast communication by only keeping the active route awake. We also present Adaptive AppSleep, an application driven addition to AppSleep which supports varying latency requirements while still maximizing energy efficiency. Our evaluation demonstrates that for an overlooked class of applications, stream-oriented power management protocols such as AppSleep outperform packet-based protocols such as B-MAC and S-MAC.

1. Introduction

Many energy efficient protocols have been developed in order to address the power management needs of wireless sensor network (WSN) applications. However, as WSN applications diversify and distinguish themselves based on bandwidth and latency requirements, no single protocol can optimally address all scenarios. For the class of latency-tolerant, non-realtime applications, leveraging an application's characteristics provides greater energy savings over traditional approaches [1] [2].

Our motivating application is Intel's FabApp [3], an application that monitors the vibration signatures of industrial equipment in a fabrication plant to predict mechanical failures. In a typical production installation, motes trans-

mit samples once a day from approximately 4000 sensors. This data is archived in a data center and analyzed offline for trends over multiple time scales. In this application, asynchronous queries typically occur shortly after pre-scheduled sample collection at the data center, and a response latency on the order of several minutes can be tolerated. Due to the lax latency constraints, nodes need not wake up with fine granularity. FabApp represents a broad class of latency tolerant applications with bursty data transmission.

Existing power management approaches do not address applications, such as FabApp, with low sampling frequencies and latency-tolerant, asynchronous traffic. These power management techniques do not take advantage of *scheduled* data transmissions or latency tolerance on *unscheduled* data transmissions to reduce power consumption. The hypothesis of this paper is that, for the class of applications represented by the FabApp, moving the power saving functionality closer to the application and tying it to transport and data collection patterns results in much higher performance.

Most power management protocols focus on ensuring that a node's radio is active for reception of each individual asynchronous packet. Thus, power management is typically tightly integrated with the MAC protocol. These protocols waste energy through radio switching, idle listening, and unnecessarily achieving short response latencies. Radio switching is especially wasteful when an application *knows* there will be no traffic. For example, if a FabApp node running on top of B-MAC with a standard sleep period of 100 ms only needs to return data once every 10 minutes, B-MAC will still turn the radio on 6000 times during that 10 minute period.

Packet-based protocols also waste energy through neighbor overhearing because they continue to wake up surrounding nodes, even during multi-fragment *unicast* streams. Using B-MAC as an example again, if a node is unicasting a multiframe data sample, neighboring nodes will wake up to hear part or all of the preamble be-

* Other names and brands may be claimed as the property of others.

fore determining the packet is not for them and going back to sleep. For long or frequent packet transmissions, or densely packed nodes, these lengthy wake up periods significantly increase energy consumption.

We introduce AppSleep, an application layer power management protocol that minimizes energy consumption for applications such as FabApp. AppSleep extends sleep periods by leveraging knowledge of pre-scheduled transmissions while still meeting the application’s latency requirements. AppSleep moves the power management functionality up the stack from the MAC in order to leverage minimal application characteristics and manage sleep schedules based on streams instead of packets. As a stream-oriented protocol, AppSleep prioritizes energy efficiency over latency, enables scheduling across multiple packets to reduce neighbor overhearing, and facilitates multi-hop communication during a single wake period. We also present Adaptive AppSleep, which handles time-varying latency requirements. Adaptive AppSleep gives applications such as FabApp the flexibility to efficiently and effectively trade latency for energy when desired.

We demonstrate that AppSleep can outperform a generic energy efficient MAC by over 3x. Our evaluation is based on both theoretical evaluation and experimental runs of B-MAC and AppSleep on a Mica2 testbed.

2. Related work

Power saving techniques have been proposed in wireless research that operate at almost every layer in the communication stack. To categorize them and present their relationship with our contributions, we develop an informal taxonomy. We define application classes (Table 1) based on traffic and latency characteristics and briefly discuss existing power management protocols designed to address these applications.

We utilize these terms in the taxonomy:

- *Synchronous Data*: Pre-scheduled sample delivery
- *Asynchronous Data*: Unscheduled data, i.e. detected events / queries
- *Synchronous Latency*: Time to return pre-scheduled data once it has been obtained
- *Asynchronous Latency*: Time to return an event once it has been detected, or the time between a query and the corresponding response.

We specify a power management protocol that best suits each class of applications. In summary:

- S-MAC and B-MAC are both designed to handle mostly asynchronous data.
- S-MAC is best suited for the *Frequent Monitoring* class of applications because it constantly expends energy to synchronize the network, which is useful only when there is a lot of traffic.
- B-MAC is best suited for the *Casual/Emergency Event Detection* classes of applications because of its low network maintenance cost. Furthermore, because communication has such a high cost when using B-MAC, it is not suitable for applications with frequent, bursty or bulk data transfers.

S-MAC conserves energy by synchronizing clusters of nodes to the same sleep/wake schedule [5]. Nodes in a cluster send frequent SYNCH packets to maintain cluster synchronization and obtain link characteristics. Nodes at cluster borders allow inter-cluster communication by operating on a super-set of cluster schedules. S-MAC has numerous drawbacks in our application. For example, S-MAC sleep schedules operate on a per-packet basis, resulting in extended multi-hop latency as nodes delay forwarding packets until their neighbor wakes up. Buffering is also required to store delayed packets.

T-MAC is based on S-MAC but seeks to eliminate idle energy further by adaptively varying the length of time over which frames are transmitted [6]. In T-MAC the data frames are sent in a burst in the beginning of the wake period and thus the nodes can go to sleep if no transmission occurs in the beginning of the wake period. Thus, T-MAC is somewhat more stream-oriented than S-MAC. In general, however, the disadvantages of T-MAC in *Infrequent Monitoring* applications are similar to S-MAC.

B-MAC is an energy efficient MAC protocol that puts nodes to sleep for tens or hundreds of milliseconds [7]. A node precedes each packet with a preamble corresponding to the maximum sleep time of neighbor nodes, ensuring that one-hop neighbors will receive the packet. The result is a significant penalty for each packet transmission. For example, for a sleeping duration of 160 ms, a node must transmit continually for 20 ms (for the packet) + 160 ms (for the preamble), adding an 8x additional overhead to the packet transmission. Furthermore, network density impacts the global energy efficiency because nodes hear some or all of each neighbor’s lengthy preamble.

Both S-MAC and B-MAC are geared towards applications with mostly asynchronous traffic. In [8], Schurgers, et al. propose a topology management protocol that puts

Table 1. Informal Application Taxonomy

Application class	Example	Min Synchronous Data Frequency	Min A/Synchronous Data Latency	Bandwidth Budget	Optimal Energy Saving
<i>Infrequent Monitoring</i>	FabApp [3]	Minutes	Several minutes	Bursty	AppSleep
<i>Frequent Monitoring</i>	Habitat Monitoring	Seconds	Several minutes	Small bursts	S-MAC
<i>Casual Event Detection</i>	Conference Room App	N/A	<= 1 minute	Small bursts	B-MAC
<i>Emergency Event Detection</i>	Fire Detection	N/A	<= 1 second	1/lifetime	B-MAC

the entire network to sleep (unlike other topology management protocols that attempt to minimize redundancy). When a node wants to send a packet it continually transmits beacons until a neighboring node wakes up. This research assumes an event tracking application model, and does not leverage application related information.

In all of these power management approaches, scheduled data transmissions consume the same energy overhead as unscheduled data. No previous power management protocol effectively addresses the *Infrequent Monitoring* class of applications. This class distinguishes itself based on frequent synchronous transmissions and tolerance for latency in asynchronous transmissions. To address these applications, AppSleep integrates the power saving function with the application layer, rather than with the MAC-layer as in the above approaches.

While application driven techniques have also been proposed in [4], this work applies to general purpose wireless networks. Sensor networks are application specific, and solutions that leverage application knowledge can often outperform general purpose approaches. AppSleep attains the benefits by leveraging limited application knowledge to maximize energy savings over other protocols for this class of applications.

Because AppSleep is stream-oriented, it can enable energy efficient multi-fragment transfers by only keeping the active route between a source and receiver awake, while all other nodes sleep. This approach minimizes the overhearing caused by packet-based protocols that repeatedly wake surrounding nodes during a single transfer between a sender and receiver. In addition, because all nodes on a path are guaranteed to be awake in AppSleep, nodes do not need to buffer packets while waiting for a next-hop neighbor to wake up as required in S-MAC or B-MAC.

3. FabApp architecture

The FabApp uses a multi-hop, cluster-based architecture, with a gateway node acting as each cluster head. The communication stack and components of the FabApp are illustrated in Figure 1.

The cluster head operates in a polling mode, creating a global data gathering schedule and initiating data transfer requests from each node. Requests typically occur daily, and each data transfer lasts a few minutes.

To start, let us assume that the cluster is in the wake state. The gateway initiates a sleep cycle by sending out a packet that puts the entire cluster to sleep for a specified duration of time (typically several hours or a day). When the nodes wake up, the gateway initiates a metric-based DSDV [13] routing protocol to establish communication paths in the network, during which, nodes within the cluster send out trace routes to the gateway. The traceroutes provide nodes with a reverse path, allowing down-stream communication for control messages. Each node joins the

cluster for which it has the lowest cost route to the cluster head. Given the size of each cluster (typically 30-100 nodes) and the routing protocol used, the cluster stabilizes in a few minutes (*route_stability_time*).

Once the routes have stabilized, the gateway polls each sensor node one at a time to initiate data gathering. The data gathering request is simply flooded into the network multiple times until the data gathering begins. When data gathering begins, nodes that are not in the data path simply go to sleep for shorter durations, as the data transfer sizes are known a priori (typically a few minutes). When the cluster head has gathered data from all nodes in a cluster, the cluster goes to deep sleep.

4. AppSleep design and operation

AppSleep is a generic protocol designed to control the sleep/wake behavior of the cluster to enable the application behavior described above. AppSleep contains components to handle sleep/wake control, time synchronization (needed for the synchronized cluster sleep/wake), bulk data transfer, SYNCH packet loss, and adaptive sleep (discussed in Section 6). Detailed design of the clustering, routing and data scheduling components are out of scope of this paper. In this section we describe the operation of AppSleep and its interaction with the other layers.

4.1. Sleep/wake control

The goal of the sleep/wake module is to ensure that all nodes in the cluster sleep and wake at the same time. This single schedule reduces idle listening and minimizes wake time. The wake period of each node has to be at least as long as it takes a packet to traverse the cluster. If a node needs to send data packets to a destination, then in addition to the wake period, the nodes should remain awake for setting up an active path to transmit data and the nodes in the active transmitting path need to be awake for the duration of the data transfer.

In AppSleep, the mechanism for the sleep/wake control relies on an initial SYNCH packet (Figure 2), flooded by

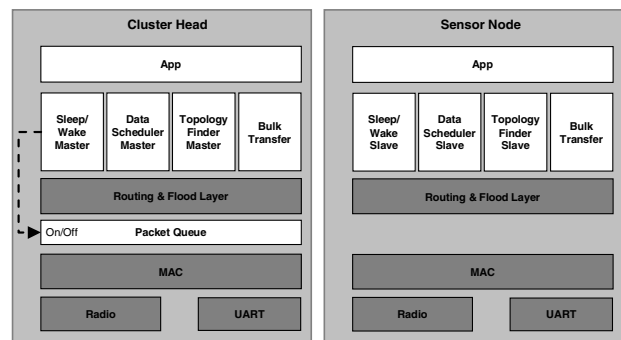


Figure 1. Cluster head and sensor node software stacks.

the cluster head. As specified in the SYNCH packet, nodes go to sleep *relative_time_to_sleep* seconds after receiving the packet, then sleep for *sleep_period* seconds. Nodes then stay awake long enough for a packet to traverse the cluster. This packet could be a SYNCH packet, or a single control or data packet. The node calculates the wake period using the *network_diameter* field in the SYNCH packet. Nodes continue this sleep-wake cycle until the next SYNCH packet is received. These parameters will be discussed in greater detail in Section 4.5.

If the payload is larger than a single packet, the bulk data transfer module is utilized (discussed in Section 4.3).

4.2. Time synchronization

A lightweight time synchronization protocol handles the synchronization between the nodes in order to ensure that all nodes wake up at the same time and stay awake long enough to hear multi-hop transmissions during a single wake period.

Related work for time synchronization [9] [10] [11], aims to maximize accuracy. While these approaches could be used, AppSleep only requires coarse grained synchronization and therefore can avoid needless overhead. The goal of AppSleep's time synchronization protocol is to minimize energy consumption while ensuring that nodes go to sleep and wake up within milliseconds of each other. In order to minimize the energy consumption, nodes use relative rather than pair-wise synchronization, sacrificing accuracy for lower packet overhead. This lower packet overhead also enables scalability, requiring on average one message per node for each time-synchronization period.

In order to achieve cluster synchronization, the protocol compensates for several factors: packet delay, packet loss, and clock drift [9]. The cluster head periodically floods a SYNCH packet containing the relative time for the cluster to go back to sleep and the length of the subsequent SYNCH period (length of time between SYNCH packet transmissions). Nodes use the relative time to go to sleep to synchronize with the cluster sleep schedule in order to compensate for accumulated clock drift since the last SYNCH packet. Nodes expect the next SYNCH packet in *time_until_SYNCH* seconds.

4.3. Bulk data transfer

AppSleep includes a bulk data transfer module to handle multi-fragment data transfers. This stream-oriented protocol puts nodes to sleep for the duration of a stream as opposed to packet-based protocols which sleep based on the contents of individual packets. Single packet data transfers can occur during the regular cluster wake period.

We begin by assuming that a route exists and is known between the sending and receiving node. A node unicasts the first fragment of the packet along this route to its des-

```
typedef struct {
    uint16_t relative_time_to_sleep;
    uint32_t time_until_SYNCH;
    uint32_t sleep_period;
    uint8_t network_diameter;
} SYNCH;
```

Figure 2. Data structure for a SYNCH packet.

tinuation, commanding the nodes along this path to stay awake by setting the WAKE bit in the packet header. The node continues to send the rest of the packets, and only sets the FIN bit in the packet header of the final fragment, indicating the nodes on the path should go back to sleep.

A timer can be used to ensure that nodes return to the sleep state if the packet with the FIN bit is lost. If a node does not forward any packets within a set time period, the node automatically returns to the sleep state. The timer must be larger than normal packet latencies but small enough to achieve the desired lifetime goal.

If the packet with the WAKE bit is lost in transit, the path will be partially awake, and the transmission will fail. To increase system reliability, the cluster wake period can be set long enough for an end-to-end acknowledgement of the initial packet and subsequent retransmissions.

4.4. SYNCH packet loss

Nodes know when to expect the next SYNCH packet because the cluster head specifies this time in each SYNCH packet. Nodes stay awake for several seconds when expecting a SYNCH packet. If a node does not receive a SYNCH packet when expected, it waits a period of time before broadcasting a SYNCH-REQ message. Any node that hears a SYNCH-REQ broadcasts an updated SYNCH packet. A node stays in this state until it hears a SYNCH packet; if it hears another node transmitting, it can broadcast another SYNCH-REQ packet. Once a node has received a SYNCH packet, it ignores subsequent SYNCH packets received during that wake period.

If a node does not hear a SYNCH packet after sending several (four in our implementation) SYNCH-REQ messages, it remains on. Nodes that join the network, or have rebooted also begin in this state. In this case, running on top of an energy efficient MAC layer can minimize the energy consumed by the radio while it waits for a SYNCH packet. This issue is discussed further in Section 7.

4.5. AppSleep parameters

There are several parameters in AppSleep that impact its energy consumption and latency characteristics.

Parameters specified/derived from the application:

- *Sleep Period* (t_{sleep}): Duration a node's radio processor is asleep before waking up

- *Network Diameter* ($N_{diameter}$): Longest path in the cluster

Parameters calculated by AppSleep:

- *Time synchronization frequency*: Frequency of SYNCH packets; optimized with respect to energy
- *Wake period* (t_{awake}): Duration a node's radio/processor is awake before going to sleep
- *Guardband* ($t_{guardband}$): Delay at packet initiator to ensure cluster is awake

The *sleep period* is set to the application's minimum tolerated response latency. The *guardband* depends on *time synchronization frequency*; as *time synchronization frequency* increases, the accrued clock drift is lessened, allowing a shorter guardband. The *wake period* depends on *time synchronization frequency* (which affects clock drift) and the cluster diameter as defined in Equation 1.

$$t_{awake} = N_{diameter} * t_{per_hop_delay} + t_{max_clock_drift} + t_{guardband} \quad (1)$$

The optimal time synchronization frequency minimizes energy consumption, based on a trade off between 1) the time the receiver is on and 2) the energy due to transmitting SYNCH packets. *Increasing* the time synchronization frequency reduces the clock drift and the *receiver on time*. *Decreasing* the SYNCH frequency reduces *transmission energy*. As seen in Figure 3, the optimal time synchronization frequency depends on the sleep period, which is determined by the latency constraints of the application.

New values for the sleep period and cluster diameter are propagated in a SYNCH packet. To allow nodes to join the edge of the cluster between SYNCH packets, a guardband of several hops is added to the cluster diameter.

5. Evaluation

Our performance evaluation of AppSleep consists of energy model evaluations and testbed experiments using an implementation of AppSleep in TinyOS on the mica2 motes. We compare AppSleep with B-MAC¹ [7] and S-MAC² [5]. While AppSleep is not a MAC protocol, we only compare the performance of the power management function of each approach. Thus, we evaluate the merit of a stream-oriented approach to power conservation.

We used network traffic that might occur in the *Infrequent Monitoring* class of applications as specified in Table 1. For streams, the energy model makes the optimistic assumption that B-MAC only uses the long preamble for the first fragment.

In both theoretical evaluation and experimental runs, an 8-node network with a fixed linear topology was used, providing a constant 7-hop network diameter. In experimental runs, nodes were placed 6 inches apart, and while a low transmit power was used, no attempt was made to

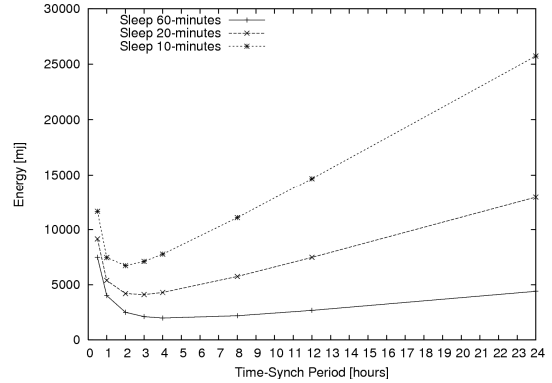


Figure 3. The energy consumed by the network as a function of SYNCH packet frequency. The data is derived from the energy model.

ensure that nodes could only receive packets from their direct linear neighbor. Data traffic was sent from one end of the network to the other at a rate of one packet every 10 min. Samples are 60 bytes long and fit in a single packet.

We first present our energy model used for the theoretical evaluation. We then discuss results from our testbed, and finally we use these results in the energy model to explore scenarios for which experiments were not feasible.

5.1. Energy model

Our energy model is based on that designed by Polastre et al. [7], and models the energy consumed by the radio by estimating the time it spends in each state. The radio is modeled as being either awake—in which case the only states are transmitting or receiving—or asleep. The model is based on a mica2 with a 19.2 kbps radio, transmission current of 20 mA, receive/idle current of 15 mA, and sleep current of 0.03 mA. The model includes control packet overhead and assumes overhearing of 5 neighbors in the topology described above. We use 5 neighbors in order to not disadvantage B-MAC, which is negatively impacted by dense neighborhoods as discussed in Subsection 5.3.3.

For AppSleep we set the *SYNCH* period to two hours. For SMAC we used a *SYNC* period of 12 sec, an awake time of 115 ms, and a duty cycle of 1%, as specified in the code. For B-MAC we used the parameters provided in [7], and validated our energy model by recreating the results offered by Polastre, et al. [7]. We validate the B-MAC and AppSleep energy models by comparing the results with those gathered in actual runs.

5.2. Experimental evaluation

To measure the energy consumption in a testbed, we measured the time the radio was on at full power, the time spent transmitting, and the number of radio state switches. The sampling period was 10 min, and AppSleep sent a

¹ tinyos-1.x/contrib/ucb/CC1000Pulse as of 9/20/2004

² tinyos-1.x/contrib/s-mac as of 9/20/2004

SYNCH packet every 2 hours. We performed two runs for each test. Figure 4 shows that B-MAC consumes more than twice as much energy as AppSleep. Surprisingly, B-MAC consumed more energy in practice than estimated by the energy model, while AppSleep’s actual energy consumed was deterministic (no visible error bars) and almost exactly equivalent to the energy model’s estimate. This behavior is expected because AppSleep is not impacted by overhearing or neighborhood density changes, while B-MAC’s energy consumption depends on the number of nodes it overhears. This comparison also holds for the time the radio spends awake as seen in Figure 5.

5.3. Theoretical evaluation

We next use the experimental results and energy model to evaluate tradeoffs between power conservation approaches. We varied the sleep periods from 20 ms to 12000 sec, kept other parameters constant (as described above), and evaluated energy consumption at different neighborhood densities, sampling rates, and network diameters. We also consider the energy required for network maintenance, radio duty cycle, and latency.

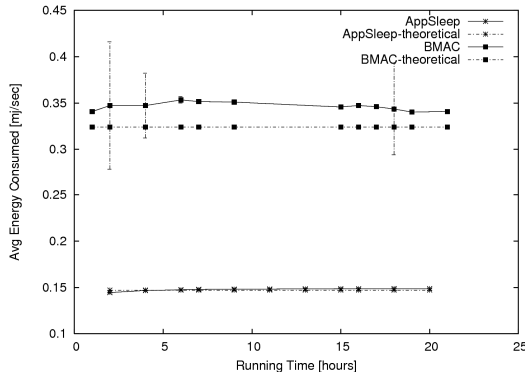


Figure 4. Experimental testbed and theoretical radio energy consumption.

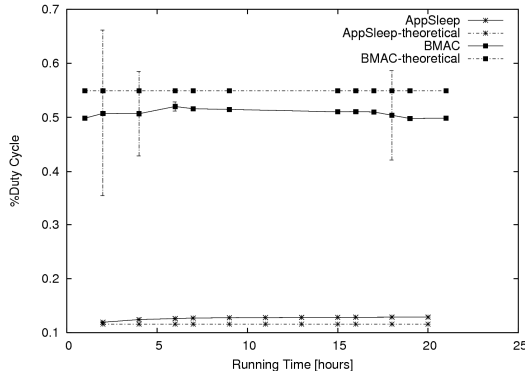


Figure 5. Experimental and theoretical radio activity.

5.3.1. Latency

The best case scenario for AppSleep is all synchronous traffic because AppSleep wakes the nodes at the previously scheduled transmission time. In this case, AppSleep has an average latency of 40 ms/hop (all transmission time), B-MAC has an average latency of 290 ms/hop if we assume an optimal sleep period of 250 ms (transmission of packet plus preamble), and S-MAC has an average latency of 2.04 sec/hop (without adaptive listening; if adaptive listening were used, the latency would be reduced).

With asynchronous traffic, the worst case scenario for AppSleep is bounded by its sleep period. In this case, B-MAC provides the best latency response (Figure 6). However, *Infrequent Monitoring* applications can often trade latency for energy efficiency, and hence AppSleep’s relatively high latency response is often acceptable.

In order to calculate average latency of response we use the following equations:

$$t_{bmac_latency} = (1.5 * t_{sleep} + t_{awake}) + (N_{hops} - 1) * (t_{sleep} + t_{awake}) \quad (2)$$

$$t_{smac_latency} = (t_{sleep}/2 + t_{awake}) + (N_{hops} - 1) * (t_{sleep} + t_{awake}) \quad (3)$$

$$t_{AppSleep_latency} = t_{sleep}/2 + t_{awake} \quad (4)$$

Where t_{sleep} is the sleep period, and t_{awake} is the period of time a node stays awake during the wake period. Notice that in B-MAC and S-MAC, each packet in a stream can incur a per-hop latency, since the next hop may be asleep. In AppSleep, after the first packet, no additional latency is incurred, since the bulk transfer mechanism ensures that all nodes along the path will be awake.

5.3.2. Sampling rate

AppSleep achieves a 3x improvement over S-MAC and B-MAC for a sampling period of 22 min, but with an average latency of 375 sec compared to SMAC’s 78 sec or B-MAC’s 77 ms (Figure 7). To surpass the lifetime of S-

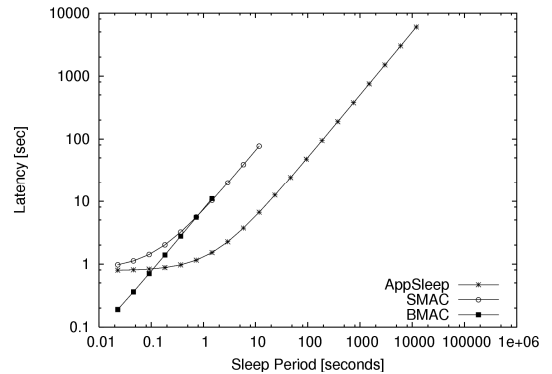


Figure 6. Average response latency as a function of sleep period. In *Infrequent Monitoring* applications, low latency is traded for energy efficiency.

MAC or B-MAC, AppSleep provides an average latency of 46 sec for the 22 min sampling period.

While the performance of B-MAC levels off, and S-MAC is implementation limited, AppSleep's performance increases with the average response latency constraint (i.e. the sleep period). As the sampling period increases to 1 day, AppSleep still outperforms B-MAC and S-MAC. However, the performance gap between B-MAC and AppSleep shrinks because this scenario minimizes transmission and is thus increasingly favorable for B-MAC.

5.3.3. Energy / latency tradeoff

AppSleep provides a tradeoff between latency and energy consumption, as shown in Figure 8. For S-MAC and B-MAC, the y-axis represents the average response latency provided at the minimal energy settings. For AppSleep, the y-axis shows the average response latency provided by AppSleep settings required to surpass the performance of B-MAC and S-MAC. While AppSleep minimizes energy consumption, it provides exponentially greater response latencies compared to B-MAC.

5.3.4. Neighborhood density

Neither S-MAC nor AppSleep is affected by neighbor density. However, B-MAC's lifetime is nearly halved when the number of neighbors is changed from 5 to 20 (Figure 9). This lifetime reduction occurs for B-MAC because as the number of neighbors increases, the amount of preamble overhearing dramatically increases.

For 20 neighbors, AppSleep shows a 9x lifetime increase over B-MAC with an average latency of 375 sec compared to B-MAC's 35 ms. Furthermore, for 20 neighbors, AppSleep surpasses B-MAC's performance with an average latency of only 5.9 sec.

Interestingly, B-MAC's optimal sleep period also changes with the number of neighbors, while AppSleep's remains constant. So, for networks with unstable links, B-MAC may not always perform in its optimal operating range, while AppSleep's optimal sleep period remains constant regardless of neighborhood density.

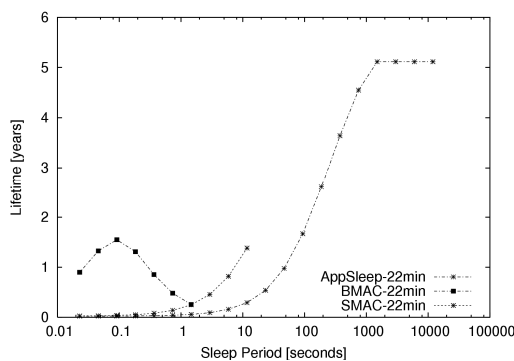


Figure 7. Theoretical lifetime as a function of sleep period (sampling period is 22 min).

5.3.5. Network diameter

As the network diameter increases, each node must stay awake for a longer period of time to ensure that multi-hop communication along the longest path can occur during a single wake period. As a result, lifetime degrades with AppSleep as the network diameter increases (Figure 10). Neither S-MAC nor B-MAC's energy consumption are impacted by the network diameter.

5.3.6. Network maintenance

Figure 11 shows the energy consumed to simply maintain network connectivity for the different protocols. When there is no traffic in the network, B-MAC consumes the least energy. This minimal energy consumption occurs because B-MAC trades reduced receiver "on" time for increased transmission time. For example, in order to consume less energy than B-MAC sleeping for 100 ms periods, AppSleep must put the application to sleep for at least 100 sec at a time. This difference occurs because AppSleep requires periodic time synchronization and, during each wake period, stays awake almost 1000 times longer than B-MAC for a 7-hop network.

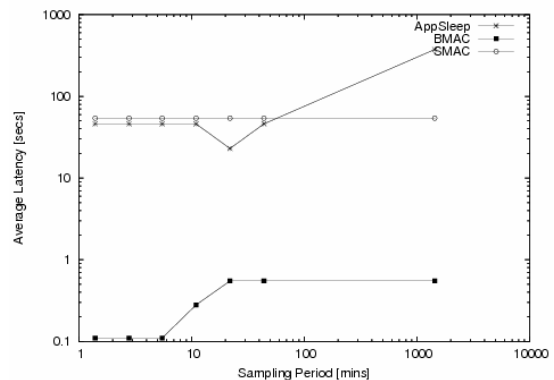


Figure 8. Response latency as a function of sampling period. Minimal energy settings used for B-MAC and S-MAC; AppSleep's parameters set to surpass B-MAC and S-MAC energy performance.

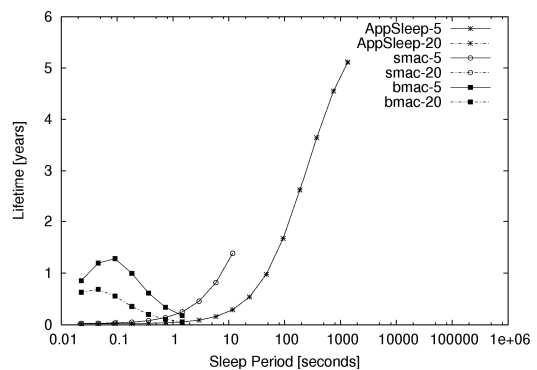


Figure 9. The theoretical lifetime as a function of the sleep period, for various neighbor densities.

6. Adaptive behavior

Adaptive AppSleep handles time varying latency constraints. It is based on the assumption that the shortest sleep periods should be scheduled immediately after scheduled sample returns because during this period unscheduled requests are most likely to happen.

Based on this assumption, Adaptive AppSleep maintains a state machine that increases the sleep period at a user specified rate after a sample has been returned. This mechanism supports the notion that latency increases in asynchronous query responses can be tolerated as time elapses since a sample return.

For example, in the FabApp application, the system returns samples once a day. If an imminent machine failure was indicated in a recent sample, a user may want to immediately query the network for additional information. However, this kind of emergency response is only necessary immediately after a sample has been returned. Subsequent requests are more likely to be informational, and can tradeoff response latency for increased energy savings.

The number of states, the time spent in each state, and the initial sleep period can be specified by the user. The cluster head handles changing response latencies and state changes by modifying the *sleep_period* field in the SYNCH message. Consider, for example, an initial sleep period of 30 sec and a duration of one hour in each state. Once a sample return is complete, the cluster head broadcasts a SYNCH packet indicating that the next SYNCH packet should be expected in one hour, and the current sleep period is 30 sec. After an hour, the cluster head floods a SYNCH packet, again indicating that the next SYNCH packet should be expected in one hour, and the current sleep period is 60 seconds. This cycle continues until the delivery of the next sample.

The state changes for Adaptive AppSleep must also be coordinated with the time synchronization protocol. With every new change in the sleep period, the cluster head

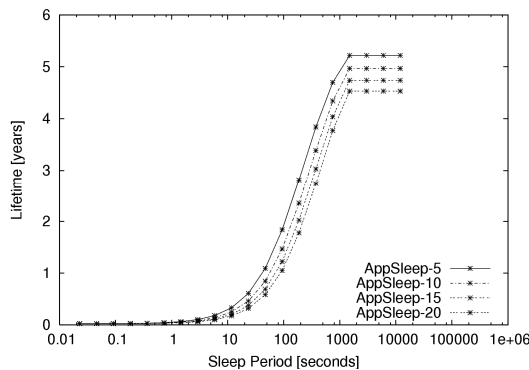


Figure 10. AppSleep’s theoretical network lifetime as a function of the sleep period, for various network diameters (sampling period is 22 min).

must recalculate its optimal time synchronization period, because this value depends on the sleep period (as shown in Figure 3). An application may utilize multiple energy-saving states, each distinguished by the sleep period.

Figure 12 demonstrates the energy consumption of AppSleep and Adaptive AppSleep. This simple implementation of Adaptive AppSleep in the energy model moves the cluster to a *query-ready* state for a period of time equivalent to 15 sampling periods immediately after a sample has been returned. For the remainder of time, nodes remain in a *minimum-power* state where their sleep periods are fixed at 10 min. The figure demonstrates the intuitive conclusion that Adaptive AppSleep saves significant energy over AppSleep by minimizing the time the network operates in the “query-ready” state and provides low latency responses to asynchronous queries.

7. Discussion

7.1. Incorporating an energy efficient MAC

There are several advantages to running this protocol on top of B-MAC. If a node fails, then it must stay awake until it hears a SYNCH packet. In this case, B-MAC is a much more efficient way to “fail on” because it will still spend most of its time sleeping and still be ensured to receive a packet from a neighbor when one is sent. Because B-MAC is a relatively cheap way to stay awake for extended periods of time, AppSleep can minimize the energy cost of extended wake periods.

To use B-MAC in this scenario, the sleep period must be very short in order to avoid large packet preambles. This approach leverages the advantage of B-MAC while avoiding the extensive switching energy usually associated with short sleeping periods in B-MAC.

7.2. Integration with and dependency on routing

Before data transfer can begin, routes must be established. To establish routes, the cluster head simply runs a

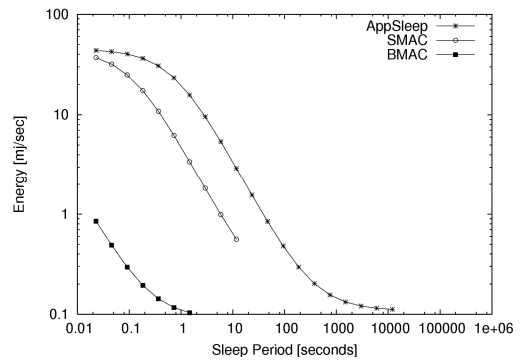


Figure 11. The power consumed to maintain the network protocol as a function of sleep period.

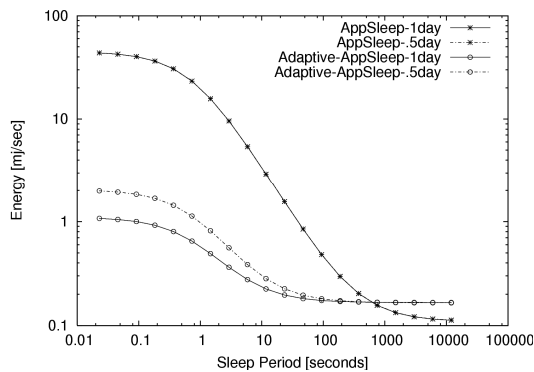


Figure 12. The theoretical lifetime of a node as a function of sleep period for two different sampling periods. The sleep period is kept at a constant of 10 min for the *minimum-power state*.

routing protocol, such as Intel’s metric-enabled DSDV [13] before initiating data transfer. Our empirical observation in networks of up to 100 nodes indicates the routes are discovered in a few minutes. The cluster head ensures that the SYNCH period supports establishment of routes.

While AppSleep avoids dependencies on the MAC layer, certain functional and performance aspects are impacted by the routing layer. Three issues must be addressed. First, during long cluster sleep periods, optimal routes may change. When this occurs, routes must be re-established at the start of the wake period.

Second, data must flow along the selected path during the entire wake period. AppSleep’s bulk data transfer protocol relies on a route remaining locked down between the time the initial WAKE bit is set requesting a route to stay awake until the final FIN packet is sent to end the transfer.

Finally an optimal routing strategy should be chosen based on AppSleep and application characteristics. For example, if AppSleep puts an application to sleep for extended periods of time, then continually updating a routing table may be a waste of energy. In this instance, a dynamic source routing protocol ideal for low traffic applications may be more appropriate. Furthermore, static timeouts that retire an entry if a node has not been heard from must take sleeping cycles into account.

8. Conclusion

We demonstrate that different energy efficient approaches address different types of applications. AppSleep is a power management protocol that addresses *Infrequent Monitoring* applications, a class previously ignored by energy efficient protocols.

AppSleep trades off latency for energy, and demonstrates greater than 3x energy savings over B-MAC and S-MAC for the *Infrequent Monitoring* class of applications. Furthermore, AppSleep is not impacted by changes in neighbor density. It also provides a solution for bulk data

transfer, ensuring that surrounding nodes do not needlessly expend energy overhearing a neighbor’s unicast transmission. In order to leverage the energy savings of this scheme, an application must be able to tolerate communication latencies for extended periods of time.

Adaptive AppSleep maximizes energy savings while handling time varying latency requirements.

References

- [1] W. Heinzelman, A. Chandrakasan, and H. Balakrishnan, “An Application-Specific Protocol Architecture for Wireless Microsensor Networks,” *IEEE Transactions on Wireless Communications*, Vol. 1, No. 4, October 2002, pp. 660-670.
- [2] P. Lerou, “Application-driven power management for mobile devices,” *Embedded Control Europe*, pp. 16–19, June 2003.
- [3] L. Krishnamurthy, J. Chhabra, N. Kushalnagar, and M. Yarvis, “Wireless Sensor Networks in Intel Fabrication Plants (poster),” Research at Intel Day, May 2004.
- [4] R. Kravets, and P. Krishnan, “Application-Driven Power Management for Mobile Communication,” *The Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, Dallas, TX, October 1998.
- [5] W. Ye, J. Heidemann, and D. Estrin, “Medium access control with coordinated, adaptive sleeping for wireless sensor networks,” *ACM/IEEE Transactions on Networking*, vol. 12, no. 3, pp. 493 – 506, June 2004.
- [6] T. van Dam and K. Langendoen, “An adaptive energy-efficient mac protocol for wireless sensor networks,” In *The ACM Conference on Embedded Networked Sensor Systems (SenSys)*, Los Angeles, CA, November 2003.
- [7] J. Polastre, J. Hill, and D. Culler, “Versatile low power media access for wireless sensor networks,” *The 2nd International Conference on Embedded Networked Sensor Systems*, Baltimore, MD, November 2004.
- [8] C. Schurgers, V. Tsiatsis, and M. Srivastava, “Stem: Topology management for energy efficient sensor networks,” *IEEE Aerospace Conference*, Big Sky, MT, March 2002.
- [9] J. Elson, L. Girod, and D. Estrin, “Fine-grained network time synchronization using reference broadcasts,” *The Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.
- [10] J. van Greunen and J. Rabaey, “Lightweight time synchronization for sensor networks,” *WSNA 2003*, San Diego, CA, September 2003.
- [11] S. Ping, “Delay measurement time synchronization for wireless sensor networks,” Tech. Rep., Intel-Research IRB-TR-03-013, June 2003.
- [12] “Digikikey cmr200t clock-crystal oscillator data-sheet,” April 2003.
- [13] M.D. Yarvis, W.S. Conner, L. Krishnamurthy, J. Chhabra, B. Elliott, and A. Mainwaring, “Real-World Experiences with an Interactive Ad Hoc Sensor Network,” *Proceedings of the International Workshop on Ad Hoc Networking*, pp. 143-151, Vancouver, BC, Canada, August, 2002.

