Adaptive Buffer Management for Efficient Code Dissemination in Multi-Application Wireless Sensor Networks

Weijia Li †, Yu Du †, Youtao Zhang †, Bruce Childers †, Ping Zhou ‡, Jun Yang ‡

†Computer Science Department, University of Pittsburgh, Pittsburgh PA 15260
‡Electric and Computer Engineering Department, University of Pittsburgh, Pittsburgh PA 15261

Abstract

Future wireless sensor networks (WSNs) are projected to run multiple applications in the same network infrastructure. While such multi-application WSNs (MA-WSNs) are economically more efficient and adapt better to the changing environments than traditional single-application WSNs, they usually require frequent code redistribution on wireless sensors, making it critical to design energy efficient post-deployment code dissemination protocols in MA-WSNs.

Different applications in MA-WSNs often share some common code segments. Therefore when there is a need to disseminate a new application from the sink node, it is possible to disseminate its shared code segments from peer sensors instead of disseminating everything from the sink node. While dissemination protocols have been proposed to handle code of each single type, it is challenging to achieve energy efficiency when the code contains both types and needs simultaneous dissemination. In this paper we utilize an adaptive buffer management approach to achieve efficient code dissemination in MA-WSNs. Our experimental results show that adaptive buffer management can reduce the completion time and the message overhead up to 10% and 20% respectively.

1 Introduction

Wireless sensor networks (WSNs) have emerged as a promising computing platform for many applications such as patient monitoring in hospitals, and wildfire detection in forests [3]. While one sensor is usually small and cheap, as the network size scales, a large WSN may contain thousands of sensors making it uneconomic to run just one application, referred as single-application WSN or SA-WSN. Instead *multi-application WSNs* (MA-WSNs) that support several applications within one WSN infrastructure are projected to become more popular [12]. As an example, a WSN deployed in a national park may be exploited for monitoring both wildfire and animals' migration habits. While a single sensor may still run one application at a time, different sensors run different applications and serve different monitoring needs.

MA-WSNs have many advantages over SA-WSNs. As an example, a MA-WSN adapts better to the changing environments as they can dynamically switch the applications according to the need. For the above WSN, more nodes can be configured for wildfire detection in the summer when the weather is dry and the chance to catch a wildfire is high; and more nodes can be configured for animals' habit monitoring either for a special short-time project or in the late fall when animals start mitigation for the winter season. As a comparison, switching applications in SA-WSNs needs to completely reprogram all nodes while reconfiguring applications in MA-WSNs only gets a subset of nodes involved.

However, post-deployment code dissemination has to be done through wireless communication that consumes significant energy. With more frequent code reconfiguration, it becomes critical to design energy efficient dissemination protocols for MA-WSNs. Our study showed that while wireless sensors cannot store all applications due to the tight storage constraints, different applications in MA-WSNs usually share some code segments. As a result, when a sensor node needs to switch to a different application, it may fetch the common code from peer sensors in the network and the rest from the sink node. While approaches have recently been proposed for disseminating each type individually, it is still challenging to disseminate code of both types of code segments. Simple solutions such as sequentially disseminating these two types of code segments, or blindly treating them as two sub applications, are not energy efficient.

In this paper we propose dissemination schemes that simultaneously disseminate the code containing both types of code segments into a subset of sensors in a MA-WSN. In particular we analyze the code propagation behaviors when disseminating code from the sink and from peer sensors, and adaptively manage the available memory on each sensor to achieve maximal energy efficiency during dissemination. We has implemented our proposed schemes in TinyOS [7] using TOSSIM [5]. Our results show that adaptive buffer management can reduce the completion time and the message overhead up to 10% and 20% respectively.

For the rest of the paper, section 2 discusses our simultaneous code dissemination schemes with adaptive buffer management. The experimental results are presented and analyzed in section 3. We will discuss the related work in section 4 and conclude the paper in section 5.

2 Adaptive Buffer Management

In this section, we elaborate the code dissemination problem in MA-WSNs and discuss different buffer management choices.

2.1 Problem Statement

The MA-WSN that we consider in this paper consists of a large number of wireless sensors and a sink node. The wireless sensors are battery-powered and have limited memory and computation power e.g. a MICAz node [11] has 4KB EEPROM data memory and 512KB flash memory. The sink node is directly connected to a PC and has no power and computation restrictions. For discussion purposes, we assume sensor nodes are pre-programmed with either application B or C and then get deployed into the field. Nodes with B and C are uniformly distributed in the network.

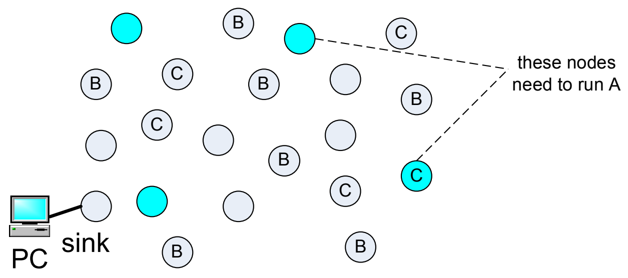

Figure 1. A multi-application WSN (A is the application to be disseminated while B and C are current applications).

The code dissemination problem in MA-WSN arises when there is a need to distribute a new application A to N_a nodes across the network. As it is infeasible to reclaim sensors after the deployment, post-deployment code dissemination has to be done through hop-by-hop wireless communication.

State	Current Draw
Radio Send state (TX, 0 dBm)	17.4 mA
Radio Receive state	19.7 mA
Radio Idle state	20μA
CPU Active mode	8 mA

Figure 2. The currents at different states on a MICAz sensor.

Fig. 2 shows the currents that a MICAz sensor draws at different states [11]. Since wireless communication is more energy expensive, it is important to reduce the number of packets exchanged during the dissemination. In addition, the processor draws current in active mode. It is beneficial to finish the code dissemination early such that they can either switch to the sleep mode or start to perform the sensing task. In summary, we evaluate the effectiveness of a code dissemination protocol in terms of a tuple (T, M) where T and M represent the time to finish reprogramming all required sensors and the number of packets transmitted during dissemination.

Our goal is to design an effective dissemination protocol such that both dissemination time and message overhead can be reduced.

2.2 Simultaneous Code Dissemination

As shown in [12], different applications in a MA-WSN usually share some code segments. For example, two applications may be designed for sensing and processing two different events — wildfire and animal mitigation. While the data processing components are different, the routing code could be similar. If one application has already been installed on some sensors, then at the time when a remote sensor wants to load the other application, it is energy efficient to fetch the common code from these peer sensors instead of the sink. Fetching code from peer sensors exhibits two advantages: (i) remote requesting sensors (i.e. the sensors that need to switch their running application to the new one) can start early and fetch the code in parallel without waiting for the progressive code dissemination from the sink. (ii) since only a subset of sensors get involved in dissemination, the message overhead can be greatly reduced. Without losing generality, we assume A and B share S_{ab} common packets while A and C do not share any packet. When a sensor needs to switch to run application A, it fetches shared code from nodes that have B and the rest of the code from the sink.

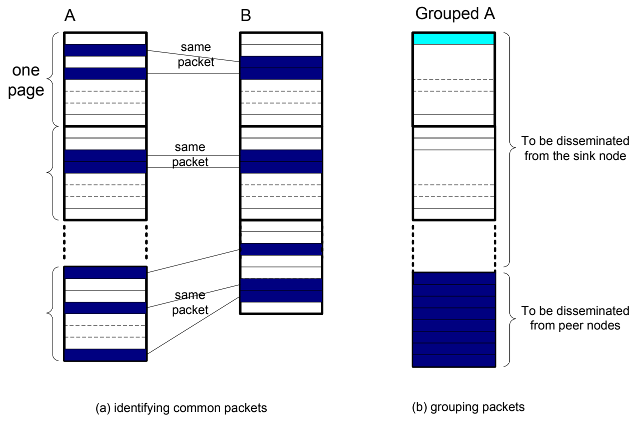
A
B
Grouped A
one page
same packet
same packet
same packet
To be disseminated from the sink node
To be disseminated from peer nodes
(a) identifying common packets
(b) grouping packets

Figure 3. Code dissemination in MA-WSN.

The basic dissemination unit in MA-WSN is a *packet* that contains 23 bytes payload, similar as that in the default multi-hop dissemination protocol Deluge [1] in TinyOS. To enable code dissemination of two types of packets, the code segments needs to be reorganized, as shown in Fig. 3. Given the above application A to be disseminated, we first divide it into a sequence of code packets, and mark all packets that are shared with B. We compare at the packet level in this paper while techniques have been proposed to compare two applications and generate difference at different levels [10, 6]. After marking the code, we group packets based on if they are marked or not, and add two bit vectors (one vector per application and one bit per packet) to guide the code reorganization. The bit vector for application A (or B) indicates the locations of marked packets in application A(or B). For example, a bit vector "011100" for A means that the 3rd, 4th, 5th packets of A are shared packets. In other words, the three packets received from the sink later on actually represent the 1st, 2nd an 6th packets of A, while the three packets received from the peer sensor represent the 3rd, 4th, and 5th packets of A.

The simultaneous code dissemination in MA-WSNs is divided into two steps.

- In the first step, the sink node broadcasts a dissemination command together with the two bit vectors, and epidemically propagates the information to all sensors in the network. This phase is fast as the data size is small comparing to the code to be disseminated, e.g. if the size of application A is 10KB or 480 packets, and assuming half of these packets are shared with B, then we need two 60-byte vectors, or 5 packets.
- In the second step, the marked packets are transmitted from peer nodes that contain B while the rest are disseminated epidemically from the sink. We adopt Deluge [1] and an augmented version of Melete [12] to handle these two types respectively. We summarize our modification as follows, and discuss the buffer management on each sensor in the next section.

Protocol augmentation. Melete adopts a controlled broadcasting design to fetch code segments from peer sensors. A requesting node broadcasts its `REQ` messages to all sensors within m hops such that all sensors containing the requested packets in this range will respond. The broadcast range can gradually extend larger if no packet was received before timeout. In practice, if there are multiple responders, message collision is a more serious problem. While Melete introduces a special response message type to prevent too many responders, the collision is still serious around the requesting nodes when there are multiple requesting nodes in the network.

In our augmented implementation, instead of broadcasting, we adopt a multi-path routing design with the help of a small table that summarizes the overheard information about each application. More specifically, the table stores (i, h) for each application X where i is the sensor ID of the closest sensor that has X (besides the node itself), and h is the number of hops to i.

This table is considered a hint and thus does not need to be accurate. To maintain the table, we enhance the heartbeat `ADV` message in Deluge which was originally designed to be sent periodically to keep the network state up-to-date. Each sensor attaches its knowledge about available applications to the `ADV` message which is used to update the table on the receiver side. Note that the table is maintained when the network is in stable state such that it has minimal interference during the dissemination. From this table a requesting node knows how far away (in terms of hop count) it can expect to find a node containing application B. Therefore the requests can be sent through one or several paths, instead of broadcasting.

If the table entry is inaccurate and the requested packets cannot be fetched before timeout, then the requesting node rolls back to broadcasting similar as that in Melete.

Simultaneous requesting both types of packets. For the two types of packets in MA-WSNs, a naive approach is to treat them as two sub-applications and fetch sequentially. However our study showed that sequential dissemination is inefficient and takes a long time to complete. Instead we prefer transmitting both types of packets simultaneously.

A requesting sensor in MA-WSNs uses different strategies for these two types: (i) for packets that can be fetched from peer sensors, it actively requests them — similar as that in Melete [12]; (ii) for packets that have to be fetched from the sink node, it waits passively until its neighbors have these packets — similar to Deluge [1]. After receiving all packets, the requesting sensor saves the code contents in the flash memory. In order to run application A, the sensor needs to get the executable, load it to its program memory, and then execute. The executable is generated by assembling the saved packets with the help of the bit vector of application A.

2.3 Adaptive Memory Management

During the code dissemination, involved sensors usually receive and buffer a sequence of code packets before taking the next action. There are two reasons.

- First, packets are usually received out of order due to lossy links in WSNs. To improve dissemination effectiveness, Deluge organizes consecutive packets into pages — the default setting is 48 packets per page. Deluge always finishes fetching the current working page before moving to the next one. When all packets in the current page are received, they are written to the flash memory.

  To support this design a sensor needs the space to buffer packets in the current page as they may be received in any order. It is energy more efficient to write at the end of receiving one page instead of each packet:
  (i) flash writing speed is slow. It takes about 78μs to finish writing one byte to the flash. As a comparison, it takes about 32μs to transmit one byte on MICAz nodes [9] ;
  (ii) flash writing consumes significant energy. It requires 3μJ and 1.5μJ to write one byte to the flash and transmit one byte respectively [9];
  (iii) flash writing has to be done at the block level e.g. 256 bytes on MICAz nodes. Since each write operation overwrites a 256 block flash, in order to change one byte in the block, the sensor has to read the correspond block, modify it in memory and then write it back. Clearly this is very energy inefficient;
  (iv) flash memory usually can sustain much smaller number of writes during its lifetime. A flash block fails after about 10,000 writes while a EEPROM block fails after 100,000 writes.
  Thus it is beneficial to reduce the number of writes to the flash during the dissemination.

- Second, different sensors usually progress differently during dissemination. As a result, a sensor often overhear packets from future pages as its nearby sensors are working on them. Since this sensor will work on those page shortly, buffering the overheard packets can reduce the total number of packets exchanged. That is , in addition to the space used to buffer the current page, a sensor may need to allocate additional space to buffer overheard packets from future pages.

Unfortunately the tight EEPROM budget (4KB on MICAz nodes) prevents free allocation of maximal page and cache sizes. If we use 48 packets per page, and separate buffers for different packet types, then we need 4416 bytes (= 2 types $\times$ (1 current page + 1 next page)/type) which is already larger than the total space.

Since the data memory is also used to store temporary variables, status information, and secret keys etc, the actual free space left for buffering code packets is usually limited. In the rest of the paper, we assume each sensor can reserve space to buffer 96 packets, or 2208 bytes (= 96 packets * 23 bytes/packet). We then study a set of different buffer management schemes and find the best one from them.

The simplest scheme is to adopt the default buffer setting i.e. use 48 packet per page and divide the available memory to two pages — one for the current page and one for the next page. Since two types of packets are transmitted in parallel, the buffer may be preemptively overtaken by each other. In the worst case the scheme may enter a deadlock if the buffers on several nearby nodes have thrashing between the two types.

To support simultaneous transmitting both types, we drop the unified buffer design and split the available memory in this paper.

- `F(24,24)` The available memory is split into two regions for disseminating two packet types independently. This scheme uses a smaller page size such that each region can still save the packets from the current page while caching the overheard packets from the next page. We set this scheme as the baseline in our experiments.

- `F(48,24)` This scheme splits the available memory similar as that in `F(24,24)`. However we use a larger page size (48 packets per page) for disseminating code from peer sensors. Since there is no space left, the packets from the next page of this type are not cached.

- `F(32,16)` This scheme splits the available memory similar as that in `F(24,24)`. However we use 32 and 16 packets per page respectively for the code disseminated from peer nodes and from the sink.

- `A(24,24)` This scheme is similar to `F(24,24)`. The difference is that when the buffer for one type is not used for a while, it can be borrowed for buffering packets of another type.

- `A(32,16)` This scheme is similar to `F(32,16)`. The difference is that when the buffer for one type is not used for a while, it can be borrowed for buffering packets of another type.

3 Experiments

3.1 Settings

We have implemented and evaluated our proposed schemes by simulating MA-WSNs of different sizes using TOSSIM [5]. We simulated 8x8, 10x10 and 12x12 mesh networks that consist of MICAz nodes running TinyOS 1.1[7]. We set the sink node at (0,0) and modeled the link failure using the $LossyBuilder$ tool with 15 feet spacing. We modified the default code dissemination protocol Deluge 2.0 [1] to incorporate the support for simultaneous code dissemination. Sensors in our MA-WSNs are pre-loaded with application B or C. We then disseminate application A to a subset of nodes in the network. We ran our experiments on a Intel Xeon 2.66GHz workstation and reported the average over 10 runs.

For the baseline setting, application A has 384 packets and needs to be disseminated to 30% nodes in the network; applicaton B was installed on 30% nodes; application A and B share 288 common packets, or 75% of application A's size.

3.2 Completion time

Fig. 4 shows the completion time for the baseline setting with different buffer management schemes. In general, schemes with adaptive buffers take less time than those with fixed buffers. For example for 10x10 network setting, `A(32,16)` takes 1186 seconds, a 7.5% improvement from `F(24,24)` which takes 1283 seconds. On average `A(32,16)` reduces the completion time by 10% comparing to `F(24,24)`.

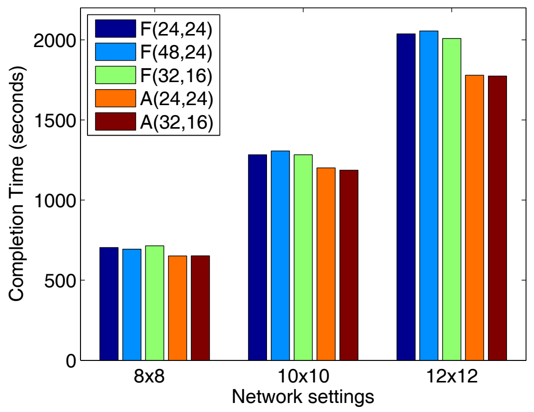

Figure 4. Completion time.

The reason for this improvement is that disseminating code from the sink node exhibits a slow start. During the warm up phase, sensors far away from the sink are not working on pages of this type and thus `A(32,16)` can devote more space for transmitting packets from peer sensors. In addition, some sensors progress faster than others, when they finish downloading, they can devote all their space for serving requests of any type. With more effective use of the memory buffer, adaptive buffer management schemes achieve faster dissemination speed.

3.3 Message overhead

Fig. 5 shows the message overhead for the baseline setting with different schemes. The results show that caching is effective in reducing the message overhead. While both `F(24,24)` and `F(48,24)` allocate the same buffer space for transmitting peer-originated code segments, `F(48,24)` dedicates this buffer to the current page and thus has no space to buffer future pages. `F(48,24)` has more packet retransmission and thus higher message overhead — on average it is about 6% more than `F(24,24)`.

Adaptive buffer management is more effective in reducing the message overhead during dissemination, e.g. `A(24,24)` gains about 10% message reduction comparing to its fixed buffer version `F(24,24)`. Out of all approaches, `A(32,16)` has the lowest message overhead — on average 20% less than `F(24,24)`.

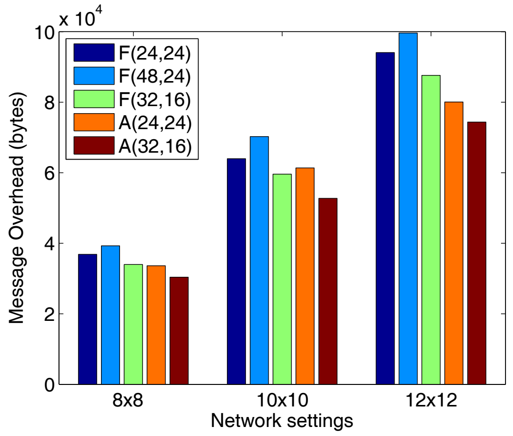

Figure 5. Message overhead.

3.4 Sensitivity

Next we study the sensitivity of our adaptive code dissemination approach under different network settings.

Fig. 6 shows the message overhead results with different number of in-network source nodes and requesting nodes e.g. (20%,10%) means 20% nodes have application B while we need to install application A to 10% nodes during the

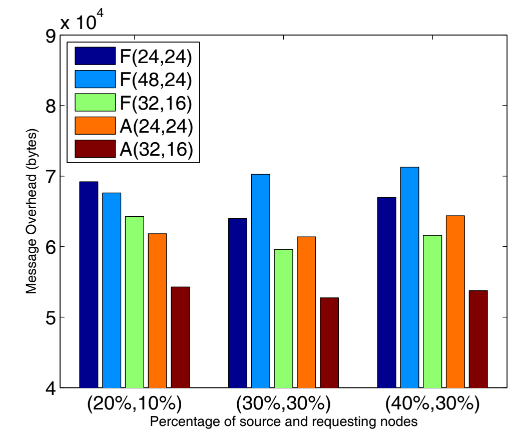

Figure 6. Message overhead with varying source and requesting nodes.

dissemiantion. As we can see from the graph, when there are more number of in-network sources, the message overhead increases due to message collision. In general we observed that our adaptive scheme `F(32,16)` shows robust performance for all settings.

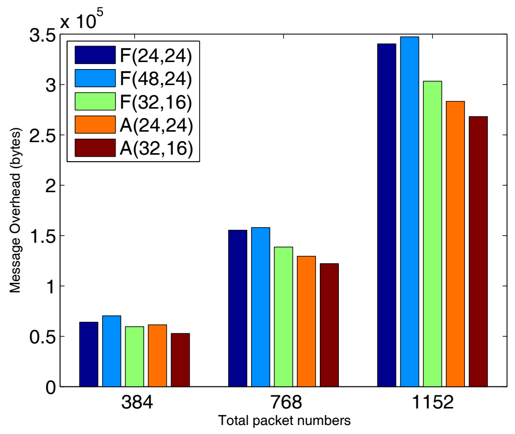

Figure 7. Message overhead with varying code sizes.

Fig. 7 shows the message overhead by varying code sizes. Clearly the message overhead increases significantly when the code to be disseminated becomes larger. When the code size doubles, the message overhead increases more than 120% on average. However, we observed almost constant message overhead reduction over all cases, indicating that the adaptive scheme is also robust with code size increase.

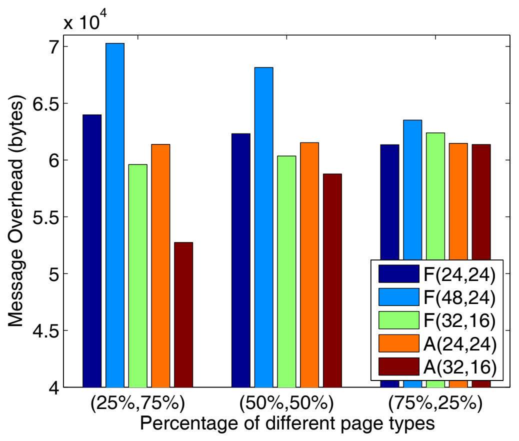

Figure 8. Message overhead with different packet distribution.

Fig. 8 shows the message overhead with different percentages of marked packets (the packets that are transmitted from peer sensors). We evaluated in a 10×10 network and the total number of packets is 384. The x-axis represents the distribution of two packet types, e.g. (25%, 75%) means 96 and 288 packets are transmitted from the sink node and peer nodes respectively. As we can see from the figure, the adaptive buffer management gains large message overhead reduction when most packets can be transmitted from peer sensors, and the benefits diminish when more packets are from the sink node. This is expected as in the latter case packets transmitted from peer nodes accounts for a smaller portion of the toal transmitted packets and thus adaptive buffer management becomes less effective.

4 Related work

Code dissemination. Several code dissemination protocols have been proposed in the literature. The ones related to our design are Deluge [1] and Melete [12]. Deluge is the default reprogramming protocol of TinyOS [7]. It assumes the unanimous application deployment in the network and employs epidemic code dissemination — the new code is first disseminated from the sink node to its 1-hop-away neighbors , and then from 1-ho-away neighbors to 2-hop-away neighbors and so on. The recent proposed Melete protocol supports multiple applications and allows in-network sources broadcasts the requested code packets to the requesting nodes. The difference between Melete and our scheme is that Melete can only fetch code from peer sensors. Most other dissemination protocols such as MNP [4] and Stream [8] assume unanimous code distribution i.e. only one application in the network.

Incremental code update. In recognizing the expensive wireless communication, several schemes have been proposed to disseminate the code difference instead of the complete code image [2, 10, 6]. When there is a need to update the existing code, these schemes focus on minimizing the code difference script such that the dissemination overhead can be reduced. Our scheme is orthogonal to these designs as we divide the code into two types and represent a more comprehensive categorization. The above designs can be considered as a special case in our design where all peer-originated packets are from the node itself and thus cause zero communication overhead.

5 Conclusions

In this paper, we study the code dissemination in MA-WSNs. By categorizing packets into two types — these that can be fetched from peer sensors and the rest that are from the sink node, we propose an code dissemination scheme with adaptive buffer management to achieve energy efficiency in MA-WSNs. Our results show that on average the adaptive scheme can reduce the completion time by 10% and the message overehead by 20%.

Acknowledgment

This work was supported partially by US NSF CAREER grant CCF 0641177 and US NSF grants CCF 0430021, CNS 0720595.

References

[1] J. W. Hui and D. Culler, "The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale," In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, pages 81-94, 2004.

[2] J. Jeong, and D. E. Culler, "Incremental Network Programming for Wireless Sensors," In *IEEE Sensor and Ad Hoc Communications and Networks (SECON)*, pages 25-33, 2004.

[3] P. Juang, H. Oki, Y. Wang , M. Martonosi, L. Peh, and D. Rubenstein, "Energy Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet," *ACM/IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 96–107, 2002.

[4] S. S. Kulkarni, and L. Wang, "MNP: Multihop Network Reprogramming Service for Sensor Networks," In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 7-16, 2005.

[5] P. Levis, N. Lee, M. Welsh, and D. Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*, 2003.

[6] W. Li, Y. Zhang, J. Yang, and J. Zheng, "UCC: Update-conscious Compilation for Energy Efficiency in Wireless Sensor Networks," In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007,

[7] TinyOS website. http://www.tinyos.net/

[8] R. K. Panta, I. Khalil, and S. Bagchi, "Stream: Low Overhead Wireless Reprogramming for Sensor Networks," In *IEEE Conference on Computer Communications (Infocom)*, 2007.

[9] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, "The Mote Revolution: Low Power Wireless Sensor Network Devices," In *HOT CHIPS*, 2004.

[10] N. Reijers, and K. Langendoen, "Efficient Code Distribution in Wireless Sensor Networks," *International Workshop on Wireless Sensor Network Architecture*, pages 60–67, 2003.

[11] MICAz Wireless Measurement System. http://www.xbow.com/.

[12] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun, "Supporting Concurrent Applications in Wireless Sensor Networks," In *Proceedings of the 1st international conference on Embedded networked sensor systems (SenSys)*, pages 139-152, 2006.