# An Adaptive Algorithm for Fault Tolerant Re-Routing in Wireless Sensor Networks

Michael Gregoire
*UMass Amherst*
mgregoir@ecs.umass.edu

Israel Koren
*UMass Amherst*
koren@ecs.umass.edu

## Abstract

*A substantial amount of research on routing in sensor networks has focused upon methods for constructing the best route, or routes, from data source to sink before sending the data. We propose an algorithm that works with this chosen route to increase the probability of data reaching the sink node in the presence of communication failures. This is done using an algorithm that watches radio activity to detect when faults occur and then takes actions at the point of failure to re-route the data through a different node without starting over on an alternative path from the source. We show that we are able to increase the percentage of data received at the source node without increasing the energy consumption of the network beyond a reasonable level.*

## 1. Introduction

In recent years companies and researchers have taken great strides towards getting to the point where we can deploy cheap, reliable and energy efficient sensor networks. One of the enablers of this progress was the advent of TinyOS [1] which presents a small yet powerful platform for developers to build sensor applications.

We propose to create a set of middleware tools to assist developers in building applications for TinyOS. Developers will supply an input file specifying which middleware services they would like and provide values for parameters that certain services will need. Some examples of services that we have worked with to this point are an RC5 based encryption solution, a message parser that reduces power usage due to header overhead and finally a fault tolerant scheme to increase the probability of successful radio transmissions in multi-hop wireless sensor networks.

In this paper we present the algorithm for fault tolerant message re-routing based on work with the TinyOS environment. The TinyOS distribution comes packaged with a multi-hop router which we refer to as "Route" as it is located in the /lib/Route/ directory of the distribution. Route establishes a tree-based network and informs each node where it stands in this network depth wise. Our algorithm, which has been written as TinyOS nesC [2] modules and tested in small mote deployments, works on top of Route in the routing layer to increase the reliability of the network. The algorithm was designed to work in an entirely distributed fashion, each node makes its decisions based solely on information it gathers by passively monitoring radio traffic around it, no feedback or direct communication with other nodes is involved. This allows configurations where some nodes run the algorithm with different parameters without interfering with other nodes.

We show here through results received from our algorithm running on the TinyViz [3] graphical simulation tool that we are able to greatly increase network reliability even in the face of high error rates in the radio environment.

## 2. Related Research

The issues that become prevalent when trying to use small and low-powered radios to form multi-hop sensor networks are well known. Not only do we have to deal with limited hardware and energy resources, but in many cases harsh environments. Many proposed deployments of sensor networks [4, 5] exhibit additional problems due to the nodes being outdoors with varying weather conditions, ground effects to nodes being close to the earth or floor and even animals destroying nodes.

A number of suggested protocols to try and deal with some of these problems exist in the literature. Some protocols have been designed specifically for usage in sensor networks and others more generally for mobile ad-hoc networks. One popular method for deciding which route to take to the sink or base station (BS) is directed diffusion (DD) [6] which begins by flooding the network and then reinforcing certain paths for later use. Other proposals have focused around Dynamic Source Routing (DSR) [7] in which the source of a data event is in charge of all the routing decisions at the time it sends its message to the sink. Another approach using light flooding when first finding the path is Distance Vector Routing (DVR) [8]. DVR is a bit different than the DSR and DD protocols as it does not attempt to find a path between source and sink until it is actually needed, saving power by not discovering unused paths. Other simpler protocols include shortest-path-first and pure flooding methods.

What these protocols have in common is that they try to form a network structure in which they determine their idea of the best path from source to sink before sending the message data. Multiple paths [9], disjoint or otherwise, are often constructed for use in case of a failure on the primary path. What we propose is an algorithm to run on top of and in conjunction with these protocols in the routing layer to help increase the percentage of data that makes it from source to sink. In our algorithm we take as given that some protocol has chosen a path it wants to use to the sink and in the case of successful routing we do not interfere with this process. However, if we notice that the next hop along the ideal path is not forwarding on the message because of either radio link or hardware issues our algorithm will attempt to find a new way to the sink from the point of failure. If we succeed we not only increase the success rate of data reaching the sink but also in many cases save power because we prevent the source node from having to try one of its pre-determined secondary paths. In this paper we demonstrate our algorithm running on top of the TinyOS Route router but it could easily be used with other path finding methods.

It is also important to discuss MAC layer protocols and their affect on the routing layer. Two popular examples of MAC layer protocols include S-MAC [10] and B-MAC [11]. S-MAC saves power by duty cycling the node's radio and waking up neighbors at the same time. It does not allow the node to listen on the radio channel or to perform retransmissions on its own. S-MAC has little interface with the routing layer. On the other side of the spectrum is B-MAC which saves power using low powered listening modes that allow nodes snoop radio traffic in the channel. B-MAC interfaces with the routing layer and depends on the routing layer to retransmit packets even if explicit ACKs are enabled. The problem is that S-MAC and B-MAC are very different in what they require from the routing layer, protocols written for one may not work for the other. For our purposes we will prefer a B-MAC or similar MAC implementation that allows the routing layer to control retransmissions. Most MAC layer protocols have some mechanism for fault tolerant retransmissions, however we believe we can increase success rates using the extra information at the routing layer.

## 3. Algorithm Description

The algorithm has been written such that any mote hardware that is supported by TinyOS is able to add the fault tolerance scheme to their TinyOS application with very little modification to their existing code. It was designed specifically for motes and hence as light-weight as possible. We also attempted to make the algorithm flexible and tunable to different application needs. While at this time we discuss TinyOS because it is the system we have implemented the algorithm for, it could certainly be easily ported to future systems.

Route broadcasts some query packets to other nodes to form a directed tree graph of nodes with the root at the BS. This tree is formed using a simple shortest-path-first methodology. Whoever a given node's parent is in the tree will forward its data on in the network until it reaches the BS. Problems can arise in this scheme when for some reason the parent is unable to forward the message. It could be that the parent node experiences a transient or even permanent failure. It could also be that another radio broadcast in the network collides with the message or just occasional data loss on a generally good radio link. In any of these cases the BS will never receive what the node had been sending its way.

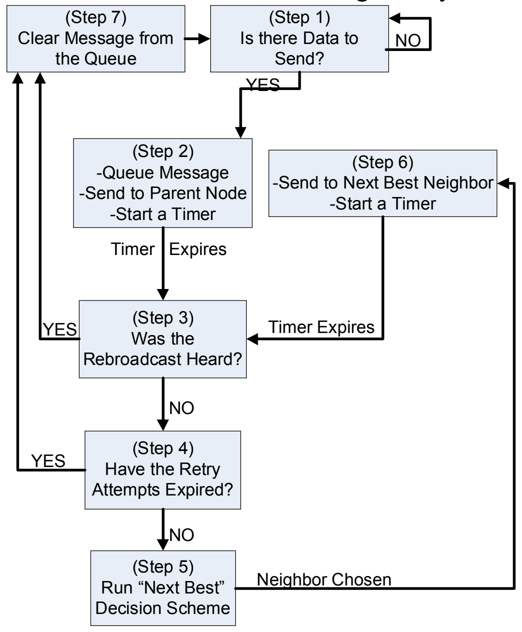

**Figure 1. A flowchart of the important steps in the fault tolerant algorithm.**

In building a fault tolerance scheme on top of Route we are given two very important pieces of information; who the node's parent is and what the depth (number of hops to the BS) of the node is within the network. By paying attention to the radio transmissions that a node can hear going on around it we can also determine who the neighbors (nodes within radio range) of the node are. We use these three pieces of information in developing the algorithm.

We have based the algorithm on what we term pseudo-ACKs. The idea is that if node A sends a message to an intermediate node B which is within close radio range, node A should be able to hear when node B sends the message to the next node C on route to the final destination (the BS). Up until the point when node A hears node B forward its message node A would continue to hold the message in queue. If enough time goes by without node A hearing a rebroadcast it will assume that there is a problem with node B and broadcast again asking a different neighbor to forward its message along. In this scheme we are using node B's rebroadcast as a pseudo-ACK. While there is no dedicated ACK packet which would affect battery life we are able to get functionality close to this by listening to the rebroadcast message that would have been sent anyways and hence add no further energy usage to the system on successful transmissions. The exception to this rule is when the messages gets one hop away from the BS, since the BS does not need to rebroadcast the message their will be no packet to use as a pseudo-ACK. In order to prevent messages that have made it all through the network from failing on their last hop we have the BS and only the BS send ACK packets for data that it receives. We do not believe this should be an issue for energy-efficiency as the BS is often a less energy limited node than the other nodes in the network. It is important to mention that pseudo-ACKs will not function in a system that uses asymmetric links. If this was the case it would be necessary to use explicit ACK packets. While we discuss the algorithm using pseudo-ACKs it could easily be simplified to work with explicit ACKs, so long as the MAC layer allows the routing layer to handle retransmissions.

In Figure 1 a full view of the algorithm is presented. Notice (step 1) that when a node has something to send it adds the message to a message queue. The length of this queue is a parameter that can be changed for different applications. If the application happens to cause a lot of traffic it might need a larger queue length. Developers may also wish to give a larger queue to nodes that are more likely to have high traffic such as those closer to the BS. A node will be able to confirm rebroadcasts of every message so long as the queue is not overrun. In the event of a queue overrun, messages that are sent while the queue is full will still be sent but will not be monitored by the fault tolerant software.

Next (step 2) the message is added to the queue and the node sends the message to its parent node (determined by Route) and starts a timer. The value of this timer is another parameter that can be set by developers and has a number of implications. If the timer length is set very high it will delay messages that require re-routing during their trip to the BS. A high timer value also means it takes longer for messages to leave the queue increasing the chance of the queue becoming full. There is also a danger in setting the timer value too small and causing retransmissions that are unnecessary. This could happen if the next-hop node is fault free and was going to retransmit the message but was busy for the timer duration. Reasons for a node remaining busy could be blocks of code that disable interrupts or a long radio queue causing the message to wait for awhile in the queue.

After the timer is started the fault tolerant software will be idle until the timer expires. During the time that the timer is running radio messages that are heard are checked against any of those in the queue checking for a match. When the timer expires the node checks (step 3) if there was a match for the message signifying that the parent received the message and is attempting to send it to the next node, this means the node needs to take no further action for this message which is then removed from the queue (step 7).

If, when the timer expires, there is no match then we check to see if there are any retry attempts left (step 4). The number of retry attempts that a node will make is the third and final tunable parameter of the algorithm. Increasing the number of retry attempts will increase the chance of messages getting through but it will also increase the overall energy usage of the network. We leave this as a parameter because some applications will care more about every event than others will. Similar to the queue length parameter it could be that nodes in certain parts of the network would be programmed with a different value for retry attempts. If all of the retry attempts have been used up, the node gives up on the message, removes it from the queue and goes back to waiting for its next message (step 7). However, if there are still retry attempts available, the node will run a "next best neighbor" selection algorithm (step 5) in order to determine which neighbor it should ask to forward the message for it. Once this scheme has chosen a node to

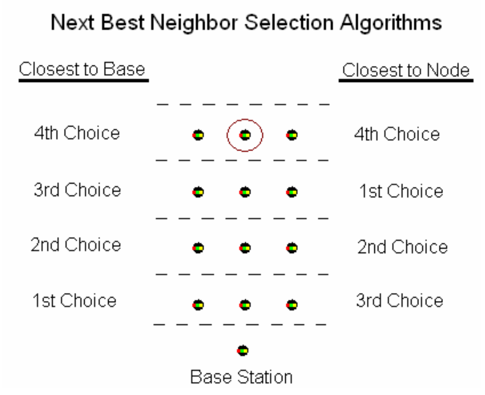

**Figure 2. Example of the two neighbor selection algorithms. The circled node is the node running the algorithm trying to choose which neighbor to send to. The dashed lines separate nodes of different depth in the network. For illustration we assume the sending node can reach and hear every node shown aside from the BS.**

re-route through it will broadcast the message to the selected neighbor and again start a timer (step 6). If the node that is asked to re-route hears the request and does forward the message on, it will do it along its own best path to the sink node as determined by Route. Just as before, the original node will monitor messages heard while the timer is running to look for a match. If a match is heard then we are done, if a match is not heard the cycle (step 4, step 5 and step 6) of checking the retry attempts, running the next best neighbor decision scheme and sending to that neighbor is repeated until a rebroadcast is finally heard or all of the retry attempts are used up.

## 4. Next Best Neighbor Selection Scheme

The next best neighbor selection scheme is an independent algorithm within the larger fault tolerant algorithm. Changing this scheme will not affect the rest of the software's operation. This is convenient because it allows us to easily test certain methods against others and allows us to use different algorithms in different applications. In this paper we examine two next best neighbor selection schemes. It is important to note that they both use the fact that nodes know the depth of their neighbors in the network through a four bit field that we have added to the header of any outgoing messages that uses our fault tolerance software. The field is loaded with the node's current depth in the network at the time of transmission. When others nodes hear the message, even if they are not the destination, they can see which node sent it and its current network depth and update it in their local table of neighbors.

### 4.1 Choose the Neighbor Closest to the BS

The simplest way to pick the next best neighbor is to look at the list of known neighbors and rank them based on their distance from the BS. This means that if node A has three neighbors, two of depth two and one of depth one then it will choose to send to the neighbor of depth one. If it happens that there are multiple neighbors that have the same depth a random number is generated to choose among these neighbors. To make sure that we are not wasting all of our tries on a node that has failed entirely we never send to the same node on two consecutive retries unless the sending node has exactly one neighbor. An example of this ranking behavior is shown on the left side of Figure 2.

### 4.2 Choose the Neighbor Closest to the Node

A safer way to pick the next best neighbor is to choose a node that is close by in the network. Since we would like whenever possible to move closer to the BS with each hop, the nodes look for neighbors that are one step closer to the BS than it is. If there is no node one step closer to the base then it looks for a node that is two steps closer to the base, continuing this until finding a node. Similar to the previous algorithm we never send to the same node on consecutive tries and break ties using a random number. An example of this ranking approach is shown on the right side of Figure 2.

The idea behind the two different schemes is that while we think that being conservative and using the neighbor closest to the node should almost always give equivalent or better rates of data transmission success we believe that in more benign environments the neighbor closest to the base method could provide similar success rates for less energy.

## 5. Results

As previously mentioned we have done small scale hardware experiments to test the validity of the algorithm. These experiments involved deploying motes with light sensors throughout a building with a BS mote attached to a laptop in one corner of the building. While we only used 12 motes this was enough to have a few nodes at depths of 1, 2, 3 and 4. When a light in a motes area was toggled on or off it would send a message to the BS laptop which had a java program listening on the serial port and would report which area of the building the light had toggled in. Using this setup we could inject faults by physically disabling motes right before toggling a light. When we ran the tests without the fault tolerance software it would often take two or three light toggles before we would actually see it at the BS, even without us injecting faults into the network. With the fault tolerant software enabled we would see it at the laptop on the first light toggle the vast majority of the time. In most cases we were also able to turn off the node's parent and see it successfully re-route the message.

In order to test our design more thoroughly we needed to employ a test bed that would allow us to produce results at a reasonable pace while still providing accuracy towards our goal of a solution that works on real mote hardware. Due to the time it takes to deploy even a small mote network we decided to gather our results using the TinyOS simulator TOSSIM [3] and its accompanying Graphical User Interface (GUI) TinyViz. This simulator gives us a good approximation of real world TinyOS applications and allows us the flexibility we need to run many different types of tests.

The simulation runs that were performed consisted of a set number of fifty nodes. We chose the number fifty because it produced results very close to those from runs with hundreds of nodes but allowed the simulator to run much faster. In all the runs nodes had a 3% chance of suffering a transient failure and becoming unresponsive for a short time. In our tests there are a number of different parameters that we set. The first two parameters are from the algorithm which was discussed previously; the number of times to retry and which of the two algorithms to use for choosing the next best neighbor. The simulator also allows us to have a simulation parameter of the network layout. The final parameter to our simulation runs is what is known as the Distance Scaling Factor (DSF). The empirical radio model that we use for our tests was created from data acquired through real mote radio tests. The model works by taking the distance between two motes and computing a bit-level error rate for a transmission between the two based on the hardware tests [3]. What this means is that by increasing the DSF we are able to keep our layout exactly the same but increase or decrease the error rate of radio transmission between nodes. We chose to use an application where every node has the same amount of data to send so that we can get a good sampling of the success rates from different depths and geographical locations in the network.

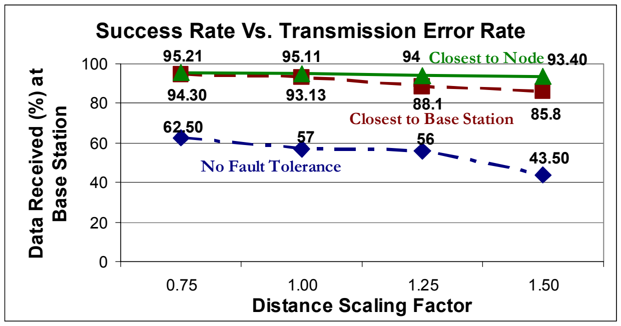

**Figure 3. A graph depicting the percentage of data that reaches the BS as the DSF or radio error rate changes.**

The first set of tests that we present involves increasing the DSF and likewise the radio error rate while keeping the layout and number of retries the same. For these tests we use 4 maximum retries and a "grid random" layout which distributes the nodes randomly about a set area. From these tests we calculate both the percentage of data that successfully arrives at the BS and the average number of radio transmissions for each message generated by a node as a measure for the energy. The results of these runs can be seen in Figures 3 and 4. We can see from these graphs that while the basic multi-hop router gives a 62.5% success rate at the lowest DSF it goes down as low as 43.5% at higher error rates. We can also see that the fault tolerant scheme provides a substantial benefit even at low error rates and becomes

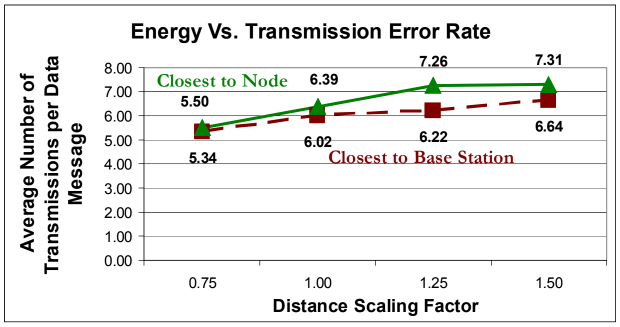

**Figure 4. A graph depicting the average number of radio transmissions sent per data message generated. This allows us to see compare energy usage between different parameter sets.**

even more advantageous at higher error rates. The fault tolerant scheme will eventually break down when the DSF starts to exceed 2.0; we do not show this on the graphs as at this point the success rate without fault tolerance is close to zero. It would appear that the Closest to Node method of choosing neighbors is better than the Closest to BS method until we look at the power graph (Figure 4). This reveals that the Closest to BS method generally uses less power.

The fault tolerant scheme is providing much more than the higher global success rate shown in Figure 3. Another benefit it provides is that its success rate holds fairly constant throughout the network. For example, the total message success rate for the data in Figure 5 without fault tolerance is 60.5%. However this is not a constant 60.5% for every node in the network, at a depth of one 70% of messages succeed while at depth four it drops to as low as 20%. This shows that while the total success rate is not so bad the BS actually barely knows anything about the half of the network that is further from it and as the size of the network expands this problem will only grow worse. We can see that with fault tolerance this problem is avoided and we have a fairly constant success rate for all depths. Figure 5 also confirms that the further out from the BS the less energy a node requires. Nodes of depth 1 use twice as much energy as nodes of depth 3. In order to deal with this the density of nodes should increase as they get closer to the BS.

The next parameter that we examine is what happens when we change the maximum retry threshold. In order to do this we again keep the layout constant throughout the tests using a random distribution within a specified area. This time we also hold the DSF (and hence the transmission error rate) constant at 1.50 and test only using the Closest to Node neighbor selection algorithm. Here we are interested in both the effect on the success rate of data reaching the BS and the energy used by the algorithm. The results are shown in Figure 6. Examining

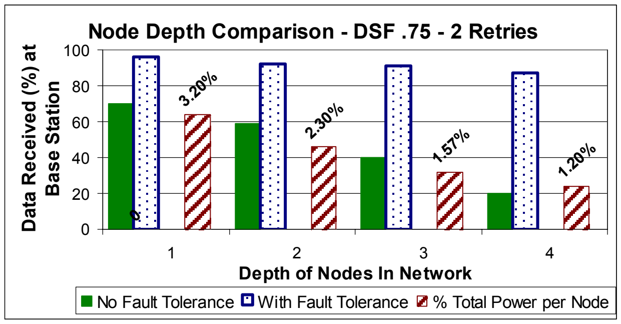

**Figure 5. A graph showing the success rates dependant on the nodes depth in the network, the dashed columns represent that percentage of overall network energy that nodes of this depth use.**

the graph we see that while there is a notable difference between 1 and 2 retry attempts, adding more retries gives diminishing returns.

In Figure 6 we present the percentage of the total network energy that is spent sending re-routing packets. This is interesting for two reasons. First, it shows that as the retry attempts go up past 2, more of the energy is being spent on re-routing messages but the overall success rate is not seeing much benefit. Second, this shows that even in a harsh environment, using 2 retries, the re-routing packets only constitute 47% of the energy in an environment where each message has a 60% chance of at *least* one error. This may sound like a large number however it is important to recall that we only send re-routing packets after a message failure has occurred. This energy usage is similar to the energy used in a multi-path scenario that takes two paths before reaching the BS.

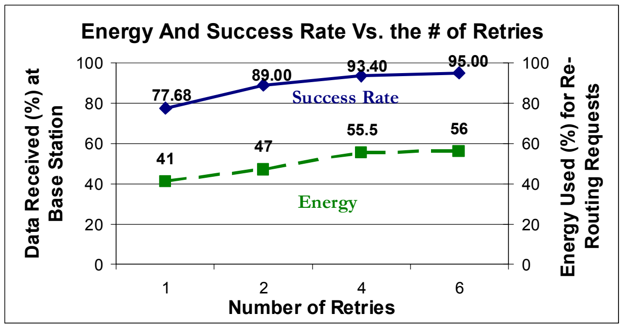

**Figure 6. A graph showing the affects of the retry threshold (left axis, solid line) on the data success rate and the percentage of energy used by the system re-routing algorithm(right axis, dashed line).**

## 6. Conclusions

We have presented a distributed algorithm for re-routing messages in the face of transmission failure in wireless sensor networks. Results were obtained from a TinyOS based mote hardware test and a number of TOSSIM simulation runs. We have seen that we can use parameters, such as the number of retries, to tune the algorithm to provide a high success rate while still being energy-efficient in both benign and hostile environments. We compared the affect on energy and success rate due to increasing the DSF. We provided results based on node depth to show that the algorithm benefits nodes further out in the network, where it is more needed, as much as those near the BS.

## 7. References

[1] TinyOS Community Forum - http://www.tinyos.net

[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC language: A holistic approach to networked embedded systems," *Proc. Programming Language Design and Implementation (PLDI)* pp. 1-11, June 2003.

[3] P. Levis, N. Lee, M. Welsh, and D. Culler. "TOSSIM: Accurate and scalable simulation of entire tinyos applications," *Proc. First ACM Conference on Embedded Networked Sensor Systems,* ACM Press, pp. 126-137, November 2003.

[4] A Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. "Wireless Sensor Networks for Habitat Monitoring," *Proc. ACM International Workshop on Wireless Sensor Networks and Applications*, pp. 88-97, September 2002.

[5] P. Juang, H. Oki, Y. Wang, M. Maronosi, L. Peh, D. Rubenstein, "Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Earth Experiences with ZebraNet," *Proc. ASPLOS*, pp. 97-107, Oct 2002.

[6] C. Intanagonwiwat, R. Govindan and D. Estrin. "Directed diffusion: A scalable and robust communication paradigm for sensor networks," *Proc. 9th Conf. on ASPLOS* , pp. 93-104, Nov 2000.

[7] D. Johnson and D. Maltz. "Dynamic source routing in ad hoc wireless networks," T. Imielinski and H. Korth, editors, *Mobile Computing*, pp. 153-181. Kluwer Academic Publishing, 1996.

[8] C. Perkins and E. Royer, "Ad-hoc On-Demand Distance Vector Routing," *Proc. 2nd IEEE Workshop on Mobile Computer Sytems and Applications*, P. 90, February 25-26 1999.

[9] D. Ganesan, R. Govindan, S. Shenker and D. Estrin, "Highly-Resilient, energy-efficient multipath routing in wireless sensor networks," *Proc. 5th annual ACM/IEEE international conference on Mobile computing and networking*, pp. 263-270, August 1999.

[10] Y. Wei, J. Heidemann, D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," *INFOCOM* vol.3, pp. 1567- 1576, 2002.

[11] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," *SenSys'04*, 2004.