January 2007

Adaptive Algorithms for Fault Tolerant Re-Routing in Wireless Sensor Networks

Michael S. Gregoire
University of Massachusetts Amherst

ADAPTIVE ALGORITHMS FOR FAULT TOLERANT RE-ROUTING IN WIRELESS SENSOR NETWORKS

A Thesis Presented

By

MICHAEL S. GREGOIRE

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

September 2007

Electrical and Computer Engineering

ADAPTIVE ALGORITHMS FOR FAULT TOLERANT RE-ROUTING IN WIRELESS SENSOR NETWORKS

A Thesis Presented

by

MICHAEL S. GREGOIRE

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 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. TinyOS has been largely accepted by both the academic and corporate communities and continues to be worked on to this day as an open source project which has just finished a milestone version 2.0 release. While there exist other operating systems that have been tested on mote hardware none of them seem to have caught on quite like TinyOS.

We have proposed to create a set of middleware tools to assist developers in building applications for TinyOS, the flow of this process is shown in Figure 1. Developers will supply an input file specifying which middleware services they would like and provide values for parameters that certain services will need along with their

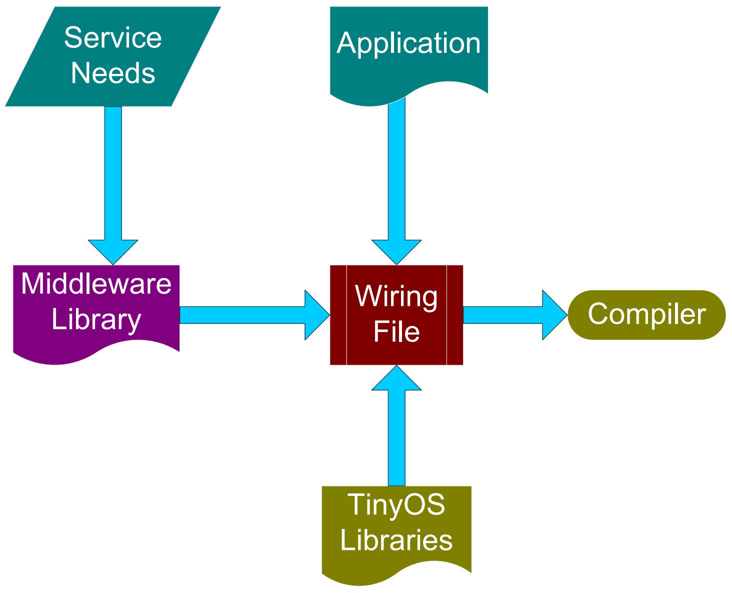

Figure 1: A flowchart of the application compiling process.

application source code files. Based on the application service needs certain middleware modules will be wired to the application in the module wiring file which connects all of the software components. This package of middleware plus application is then sent to the compiler and programmed onto the mote.

One example of a service that we have worked on is an RC5 based encryption solution that can modify the encryption strength and the key being used on the fly. This is a step forward from past work [20] that focused around similar but static encryption mechanisms. Work was done on a message parser that reduces energy by saving on header overhead and finally the most time has been spent on a fault tolerant scheme to increase the probability of successful radio transmissions in multi-hop wireless sensor networks. This fault tolerant scheme and its many different flavors and settings is the focus of this thesis.

We begin by presenting 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 "Route" that establishes a tree-based network and informs each node where it stands in this network depth wise. While the router does a good job of forming a network routing structure it does not support retransmission of messages in the case of failures. This creates an issue as the TinyOS MAC layer depends on the higher layers to retransmit for it and will not do this on its own. This leads to a situation where developers must handle all of the retransmissions within every application that they write. Our algorithm, which has been written as TinyOS nesC [2] modules and tested in small mote deployments, sits on top of the TinyOS radio stack in the routing layer and builds upon this existing structure to provide retransmissions and increase the reliability of the network.   We chose to work on top of the existing routing layer based on our middleware-centric approach. By implementing just above the routing layer we are not modifying any existing TinyOS modules and are able to have simple interaction with the developers' application. 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.

Once the groundwork for the routing layer algorithm is complete we discuss a number of more interesting extensions to the core algorithm behavior. This includes adding learning, i.e. the ability for nodes to remember past behavior of their neighbors. We also look at the affects of correlated events in a network. Correlated events are important because in a real world scenario events are very likely to be tightly coupled and all of the traffic in one area of the network can cause degraded performance due to many packet collisions and queues filling up. We discuss a number of different mechanisms for dealing with such a scenario. We also look into other interesting scenarios such as what happens when we switch the routing layer to another TinyOS compatible routing layer, the effects of duty cycling on the algorithm as well as what happens at different depths of the network as the number of nodes expands further and further.

CHAPTER 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 outdoor environments as shown in Figure 2 from [21]. 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 initially for usage in sensor networks while others where originally proposed for general mobile ad-hoc networks and later suggested for use in sensor networks. There are two basic groups that all of the proposed routing algorithms fall into, proactive and reactive. We now summarize some of the more popular routing algorithms.

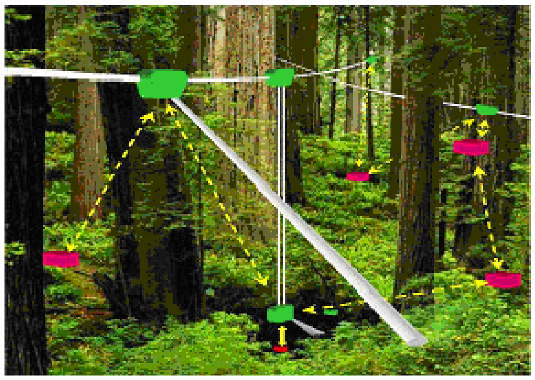

Figure 2: Examples of outdoor sensor network distributions depicting the difficult environments in which nodes need to work. On the left nodes in trees and on the forest floor monitor the habitat, on the right a node is hung from a wire fifty feet above ground.

Proactive algorithms are those which find all of the paths to other nodes ahead of time and store them in a routing table in memory. The protocols aim to update the routing table in a reasonable amount of time when there is a change in the network topology (new nodes starting up or old nodes failing). The most popular example of this is the Destination Sequenced Distance Vector (DSDV) protocol [13]. DSDV works by trying to find the shortest route (in hops) from every node to every other node while of course avoiding loops. In DSDV each node periodically broadcasts its view of the network so that its neighbors can see it and modify (or not) their view accordingly. In Figure 3 taken from [8] we see how DSDV works. On the left hand side we see the source node, s, sends a broadcast packets (depicted by a dashed circle) to see who its neighbors are. Each neighbor will do the same and responds with their distance from the source (shown as a solid arrow). Eventually the destination, d, will hear a broadcast packet and the process is complete. The forward path as shown on the right hand side is chosen based on whichever is the shortest number of hops; in this case there is only one possible path which is shown in dashed lines. The two packets that timeout do so because they have no path to the destination.

Another proactive algorithm is the Optimized Link State Routing (OLSR) protocol [14]. Like DSDV nodes in OLSR will occasionally send out messages updating their view of the network topology. This technique is known as “light” flooding because nodes periodically flood the network with their information but are not constantly flooding. The major difference between DSDV and OLSR is that in OLSR the primary information is based on the cost of transmitting on links rather than solely on who the neighbors are. The cost of the links are used to find a lowest cost path between every two

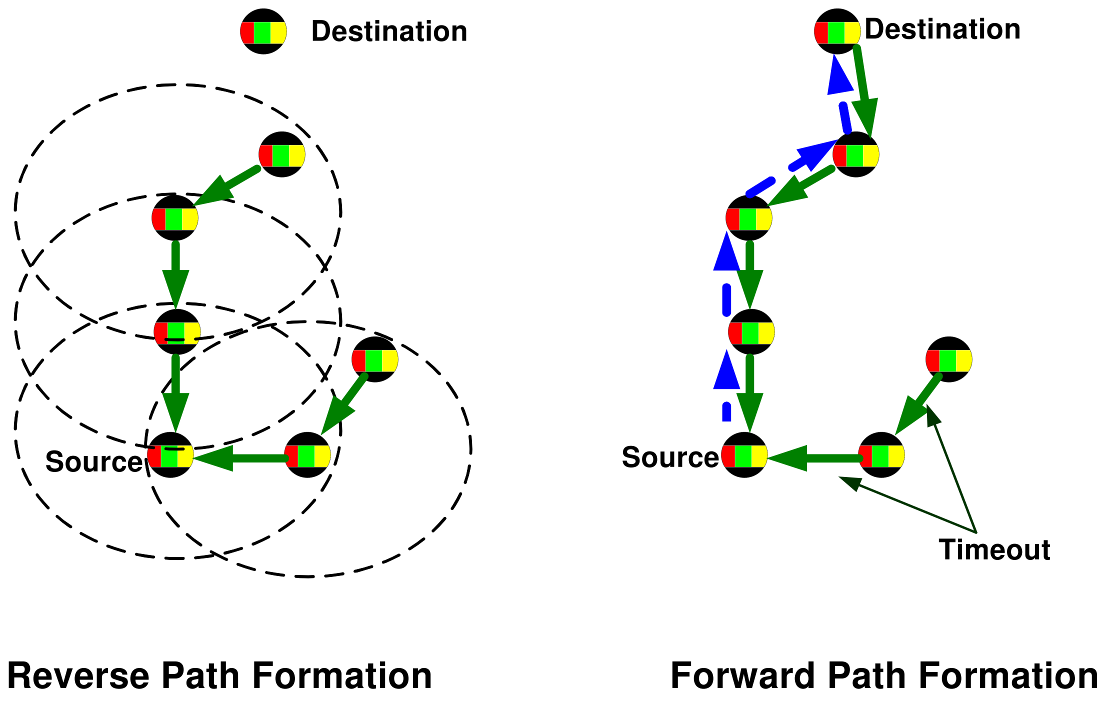

Figure 3: The general behavior of the DSDV algorithm. [8]

nodes in the network and stored in the routing tables. Source Tree Routing (STAR) in [16] is a similar but more energy efficient approach to routing-table based protocols. It uses link information like OLSR but attempts to save energy by not requiring every node to know as much information about the overarching network topology. In STAR, nodes send out information about their preferred links, and nodes form a "source tree" which is constructed from this information.

Reactive algorithms are those which only find the path that they will take to the node at the point where they need that path. Some of these algorithms start the route discovery procedure from the beginning of the path while others do it from the end of the path. Once a path has been discovered it is usually kept up as long as it is used occasionally and does not have node failures. One reactive algorithm called Ad-Hoc On-Demand Distance Vector Routing (AODV) [8] is based on the same ideas as the proactive DSDV algorithm except that in this case it is reactive. AODV will discover paths starting at the source node. The discovery process works as follows: the source node broadcasts a route discovery packet which continues to be broadcast by neighbor nodes until the request finds its way to the destination or one intermediate node that hears the request already has a path to the destination. Once the path is established the packet is sent down this path to the destination. Maintenance will take place occasionally if the path is still seen as useful.

Directed Diffusion (DD) [5, 19] is another reactive protocol which was originally designed with sensor networks in mind and sets itself apart from other protocols because it is data-centric and application aware. In directed diffusion all data generated by sensor nodes comes in an attribute-value pair. When one node has interest in a certain type of data it will send out the request through the network. If the request reaches a node that has relevant information it will send that information to the node that sent the request. In this way DD starts the path discovery at the node which is the destination of the data. DD also has the feature of combining two packets into one packet if they are both carrying the same type of information. Other reactive protocols include Dynamic Source Routing (DSR) [7] , Temporally-Ordered Routing Algorithm (TORA) [15] and Energy Aware Routing [10]. DSR is similar to AODV except that in DSR all of the nodes eventual routing information (rather than just the information for this hop) has to be provided by the source node before the packet is sent to the destination. TORA is based on what is called a “temporal clock” which places an order on any topographical change that takes place in the network. When these changes happen, TORA runs its distributed algorithm in order to replace the paths that were lost. In energy aware routing nodes

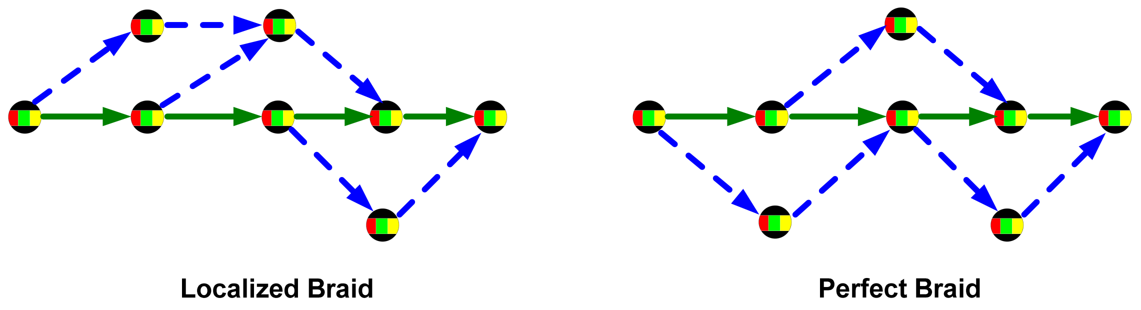

Figure 4: An example of braided multipaths. The sold lines represent the primary path and the dashed lines represent alternative paths [9].

sometimes use sub-optimal paths so as to better distribute the load throughout the network and increase the lifetime of the entire network.

What these protocols have in common is that they try to form a network structure in which they determine a path from a source node to the base station before sending the message but do not always have a fall back, fault tolerant, plan. Multiple paths as in [9] are often constructed for use in case of a failure on the primary path, an example of one multipath scheme from [9] is shown in Figure 4. The idea of braided multipaths is to have alternate paths that allow a message to leave the primary path in the event of a path failure. On the right the perfect braid is shown, this is the best case scenario in which there is a path to skip any node on the primary path without increasing hop length. On the left is the localized braid which is formed using a more practical algorithm. 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 base station. In our algorithm we take as given that some protocol has chosen a path it wants to use to the base station 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 base station from the point of failure. If we succeed we not only increase the success rate of data reaching the base station but also in many cases save energy as opposed to a multipath solution because we prevent the source node from having to try one of its pre-determined secondary paths. In this thesis we demonstrate our algorithm predominately running on top of the TinyOS "Route" multi-hop router but also show an example of how it can work with any routing layer that implements the standard TinyOS routing interfaces without modifying a single line of source code.

Another important aspect that should be discussed is the MAC layer and how it interacts with the routing layer. The two most popular MAC layers that have been proposed and implemented for TinyOS are S-MAC [17] and B-MAC [12]. While they have similar goals and are both based on trying to avoid packet collisions their implementations are quite different. B-MAC saves power by having eight different low-power listening modes which adjust the preamble to a lower value in order to save more power. B-MAC allows every node to overhear every packet in its radio range and pass them to higher layer protocols. B-MAC has the option of enabling ACK packets but if this is done B-MAC assumes that the retransmissions will be carried out by some higher (routing) layer. In [11] the authors discuss how to appropriately set the listening-level in B-MAC for use in existing TinyOS routing layer protocols. S-MAC does not use low power listening but instead turns the radio off periodically. Nearby neighbors are kept on the same schedule of when to be awake and when to be asleep so that packets can be heard between them. The problem as discussed in [18] is that B-MAC and S-MAC both provide different information and have different expectations from the routing layer. B-

MAC has parameters which can be modified by upper layers and provides them with all overheard traffic while S-MAC can not be modified by higher level layers and does not provide them with overheard traffic; in fact it tries to avoid hearing as much traffic as possible. This means that when developers are writing applications they can not pick a MAC layer and routing layer independently as protocols for one will not work with the other. For our purposes we prefer B-MAC because it allows the routing protocol to choose whom to retransmit to in the face of failure and allows nodes to snoop on radio traffic.

CHAPTER 3

BASIC ALGORITHM

3.1 Approach and Description

The re-routing algorithm has been written such that any mote hardware platform that is supported by TinyOS is able to add the fault tolerance scheme to their TinyOS applications with very little modification to the existing application 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.

The TinyOS multi-hop router broadcasts some query packets to other nodes to form a directed tree graph of nodes with the root at the base station. 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 base station. An example of this can be seen by comparing Figure 5 to Figure 6. Figure 5 shows an

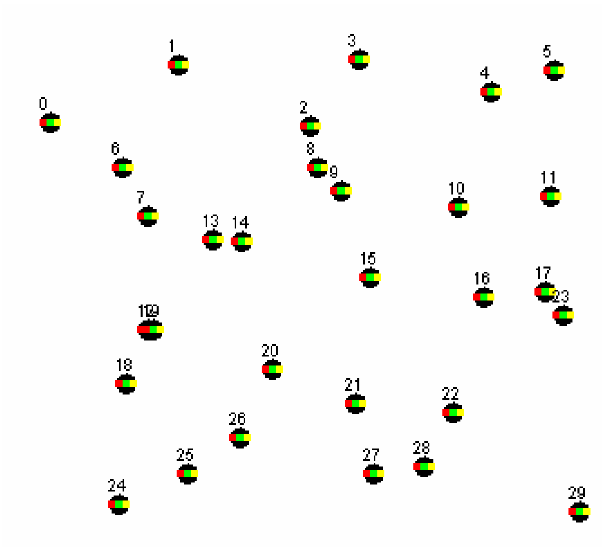

Figure 5: The geographical layout of a network.

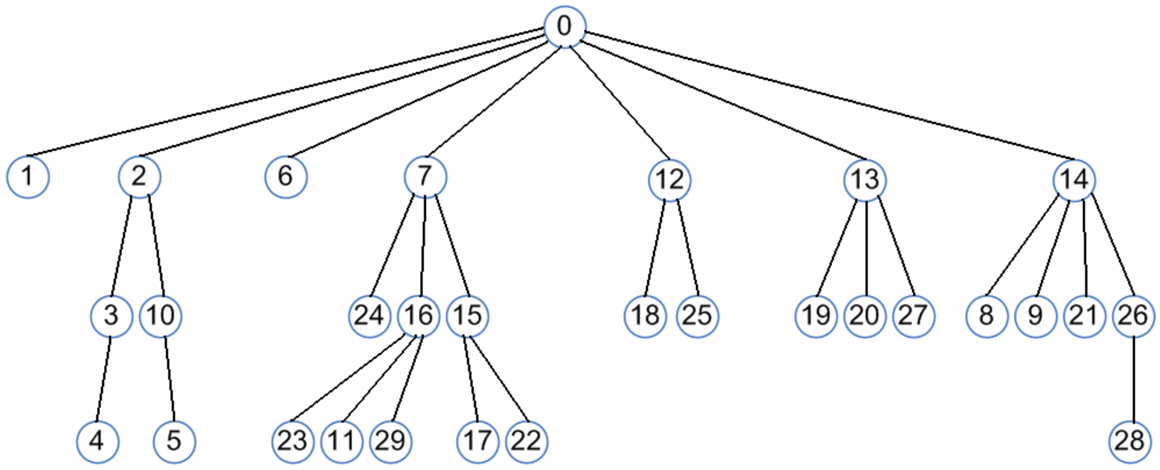

Figure 6: The routing tree formed by the TinyOS multi-hop router.

example node geographic layout that was run in the simulator while Figure 6 shows the directed tree graph that is formed from this layout by the TinyOS multi-hop router. It is clear that this is not an optimal scheme in terms of energy as certain nodes have many more children (and grand-children) then others. As mentioned in the related research section a good deal of work is being spent on different algorithms to approach this problem.

Problems can arise in the TinyOS (or any alternative) routing scheme when for some reason the pre-determined parent node 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 base station will never receive what the node had been sending its way due to the lack of retransmissions occurring in the existing routing layer. A simple example of what we would like to achieve with re-routing is shown in Figure 7. In this figure we see that when a node (number 7) receives a new packet to send, it will always initially ask its router-determined parent (node 8) to forward the message along (labeled A). This is to

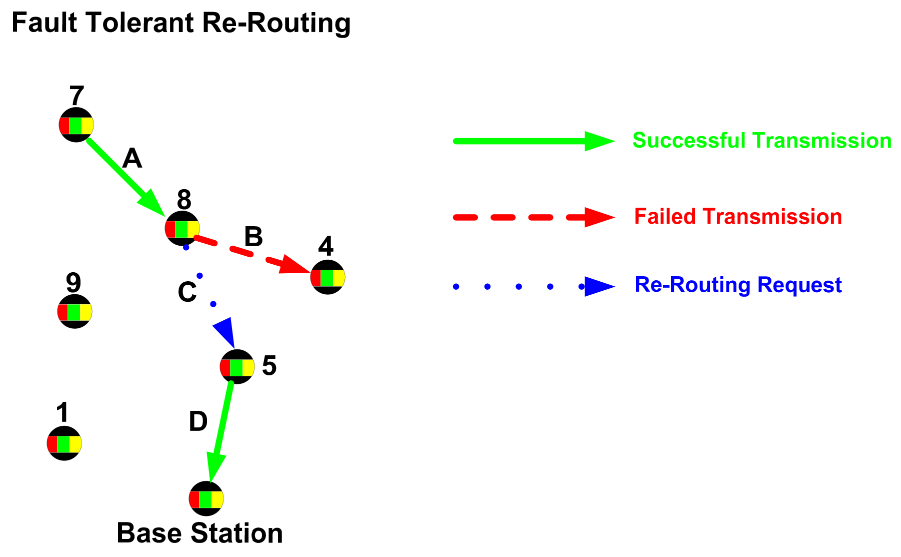

Figure 7: An example of fault tolerant re-routing.

preserve our goal of only interfering when it is necessary and otherwise allowing the router's decisions to run their course. In the next hop (labeled B) the situation arises where the router-determined parent (node 4) fails to forward along the message, only then does the fault tolerant software step in and make a decision about what to do next in order to get the message to the base station. In the case of this example that decision was to ask another node, number 5, to forward the data along (labeled C). The message then continues along the router determines path until reaching the base station (labeled D).

In building a fault tolerant scheme on top of this basic router we are given two very important pieces of information; who the node's parent is and what the depth (number of hops to the base station) 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. Although this may seem like a limited amount of information we will see that it can provide the basic information required for re-routing decisions.

Throughout this thesis we will be using the concept of an implicit ACK. The main 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 base station). Up until the point when node A hears node B forward its message node A would continue to hold the message in a queue. If enough time goes by without node A hearing a rebroadcast it will assume that there is a problem with node B and will broadcast again asking a different neighbor to forward its message along. In this example we are using node B's rebroadcast as an implicit 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 base station; since the base station does not need to rebroadcast the message there will be no packet to use as a pseudo-ACK. In order to prevent messages that have made it all the way through the network from failing on their last hop we have the base station and only the base station send explicit ACK packets for data that it receives. We do not believe this should be an issue for energy-efficiency as the base station is often a less energy limited node than the other nodes in the network. It is important to note that our scheme could easily be used with explicit (separate) ACK packets but as our radio models will use generally symmetrical radios and our MAC layer allows us to overhear neighbor's messages we believe that it is appropriate to try and save energy by skipping these explicit ACK packets.

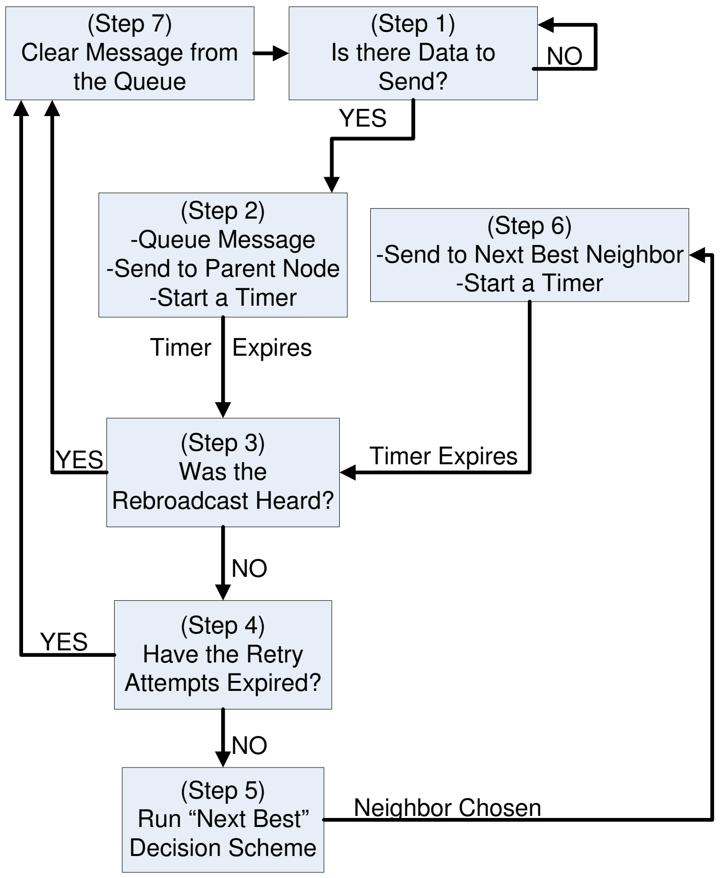
(Step 7) Clear Message from the Queue
(Step 1) Is there Data to Send?
NO
YES
(Step 2) -Queue Message -Send to Parent Node -Start a Timer
(Step 6) -Send to Next Best Neighbor -Start a Timer
Timer Expires
Timer Expires
YES
(Step 3) Was the Rebroadcast Heard?
NO
YES
(Step 4) Have the Retry Attempts Expired?
NO
(Step 5) Run "Next Best" Decision Scheme
Neighbor Chosen

Figure 8: A flowchart of the important steps in the fault tolerant algorithm.

In Figure 8 a full view of the algorithm is presented. Notice that when a node has something to send, it adds the message to a message queue (step 1). The length of this queue is the first 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 base station. 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.

In step 2 the node sends the message to its parent node (determined by the underlying TinyOS multi-hop router) and starts a timer that sets the length of time that the node will wait to hear the message rebroadcast. 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 base station more than is necessary. 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 to see if there is a match, if there is a match a flag is set on the queue slot saying that the message was heard. When the timer expires the node checks (step 3) if a match for the message has been heard to signify that the parent received the message and is attempting to send it to the next node, if this is the case 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 has been no match, 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 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, it takes over the responsibility for the message and attempts to send it along its own pre-determined (by the TinyOS multi-hop router) best path to the base station. 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.

3.2. Next Best Neighbor Selection Scheme

The next best neighbor selection scheme is a simple, independent algorithm within the 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 also allows us to use different algorithms in different applications. In this section we examine two static next best neighbor selection schemes.

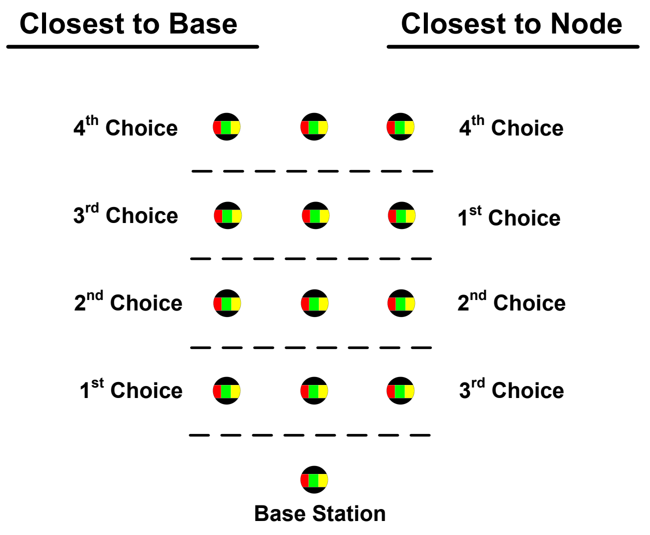

Figure 9: The next best neighbor selection algorithms. The circled node is the one which is running the algorithm and each dashed line represents a depth increase in the routing tree. The rankings of nodes of a given depth with each selection algorithm is shown

We term them static because they do not take into account any of their past successes or failures when making a decision. 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 parent, they can see which node sent it and its current network depth and update it in their local table of neighbors.

3.2.1 Choose the Neighbor that is Closest to the Base Station

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 base station. 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 attempts 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 9 and Figure 10.

3.2.2 Choose the Neighbor that is 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 base station with each hop, the node looks for neighbors that are one step closer to the base station 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 9 and Figure 10.

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 transmission success rates we believe that in more benign environments the neighbor closest to the base method could provide similar success rates for less energy. This is because we are able to reduce the number of hops that a message has to take on its path from source to base station. The closest to base method may also benefit from the fact that it is reducing the number of points of failure as opposed to the closest to node method.

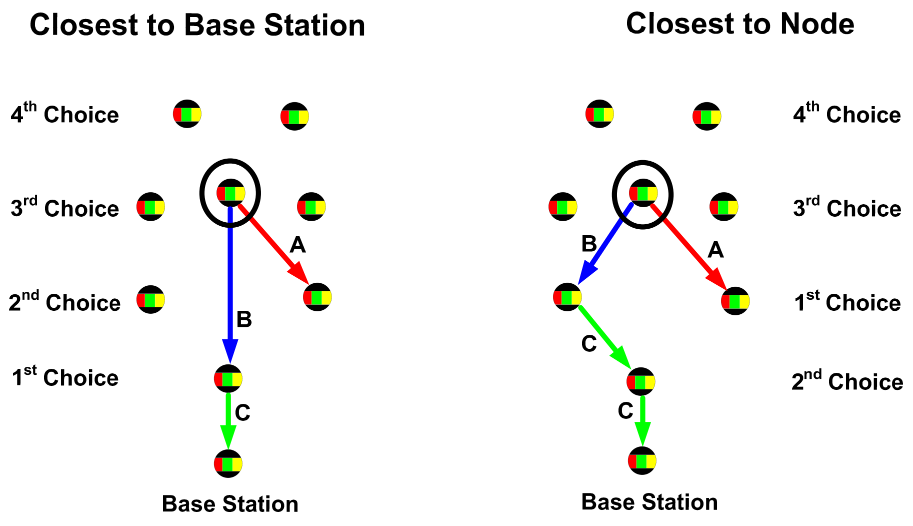

Figure 10: The two different approaches to choosing the next best neighbor. The circled node attempts to send a message (A) to its parent node but the transmission fails. It then runs the next best neighbor selection scheme and re-routes the message through another node (B).

3.3 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 base station mote attached to a laptop in one corner of the building. While we only used twelve motes this was enough to have a few nodes at depths of one, two, three and four. When a light in a motes area was toggled on or off it would send a message to the base station laptop which had a java program listening on the serial port and would report which area of the building the light had toggled. 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 base station, even with no faults injected 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

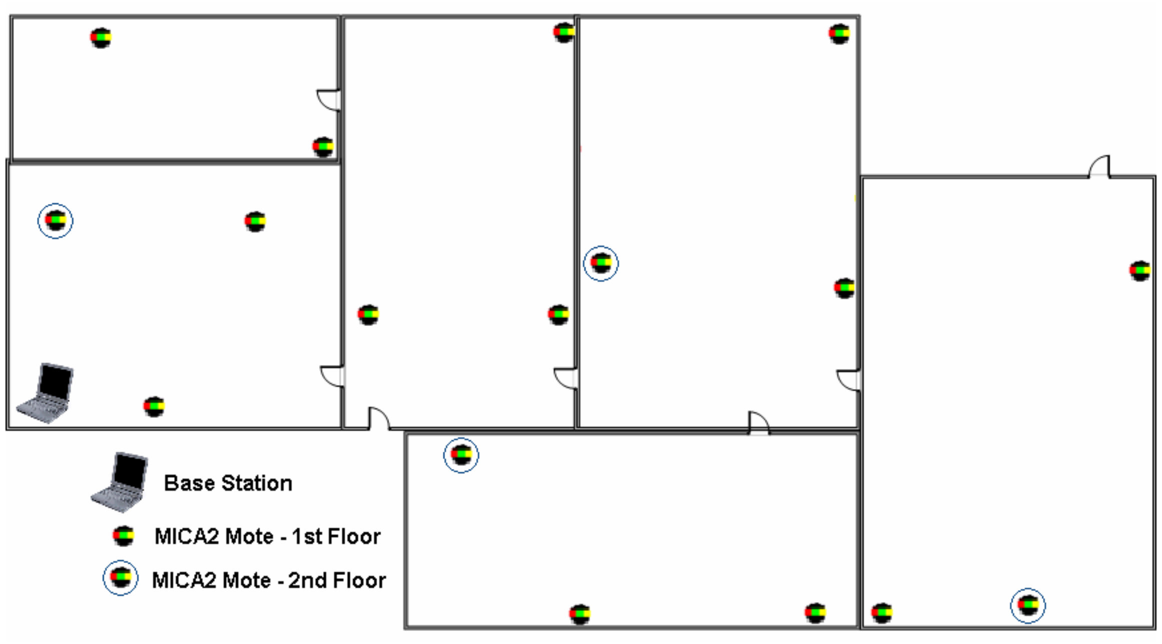

Figure 11: An approximate layout of the hardware experiment performed to verify software functionality. The test actually took places on two floors with this same room layout. The nodes that were on the second floor are depicted with circles.

also able to turn off the node's parent and see it successfully re-route the message. An approximate layout for this hardware experiment is shown in Figure 11.

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. The problem with actual hardware tests is the time it takes to deploy even a small mote network and test that the radio links are working. This is exacerbated by placing the nodes in the same location each time and attempting to run the experiment before realizing there is a minor code bug and all the nodes must be collected, reprogrammed and redeployed. As a solution to this 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.

While there are a number of more advanced general-purpose network simulators, TOSSIM is a good choice because it allows us to run the simulations using the same TinyOS code modules that we would then use on the real hardware. In order to build a TOSSIM input you take the same code that you have been compiling for the hardware and compile it with a different flag. This means that if the code works in TOSSIM you have moderate assurance that it will work in your real motes. While there may be some small timing issues that crop up due to idealizations in the simulator, the core functionality is exactly the same. This is important for our work as we want to have a system which would be implemented in TinyOS software as opposed to some other more popular language with the claim that it could be implemented for TinyOS. Writing TinyOS modules also allowed us to be well aware of exactly what functionality is provided by the operating system and what the limitations were with both it and the mote hardware.

The simulation runs that were performed usually 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. The time it takes to run a simulation increases at a superlinear rate meaning for example that running a one hundred node simulation would take four or five times as long as a fifty node simulation. In all of the simulations we will have a certain probability of nodes experiencing a transient failure preventing them from sending or receiving messages from other nodes for a varying period of time. On top of this we use a lossy radio model built into TOSSIM to simulate radio collisions and bit level errors in packet transmissions.

In our tests there are a number of different parameters that we commonly set. The first two parameters are for the algorithm which was discussed previously; the number of times to retry and the way that we choose who the next best neighbor to send to is. The simulator also allows us to have a simulation parameter of the network layout. The final parameter in our simulation runs is what is known as the Distance Scaling Factor (DSF). The empirical radio model used in TOSSIM is a lossy radio model that provides bit level error rates on transmitted packets. The error rates that it uses come 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

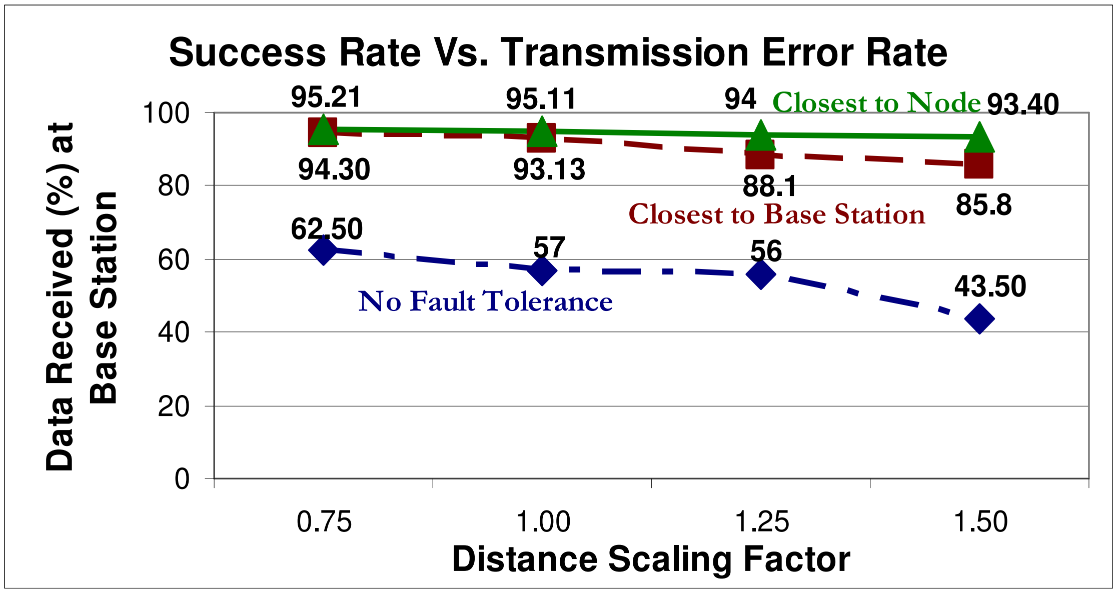

Figure 12: The percentage of data that reaches the base station as the DSF changes.

between nodes. Therefore, the DSF is essentially an environment factor, if we keep the layout the same and increase the DSF we increase the success rate of transmissions between every pair of nodes in the network.

The first set of tests that we present involves increasing the DSF and the radio error rate while keeping the layout and number of retries the same. For these tests we use four 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 base station 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 12 and 13. 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 even more advantageous at higher

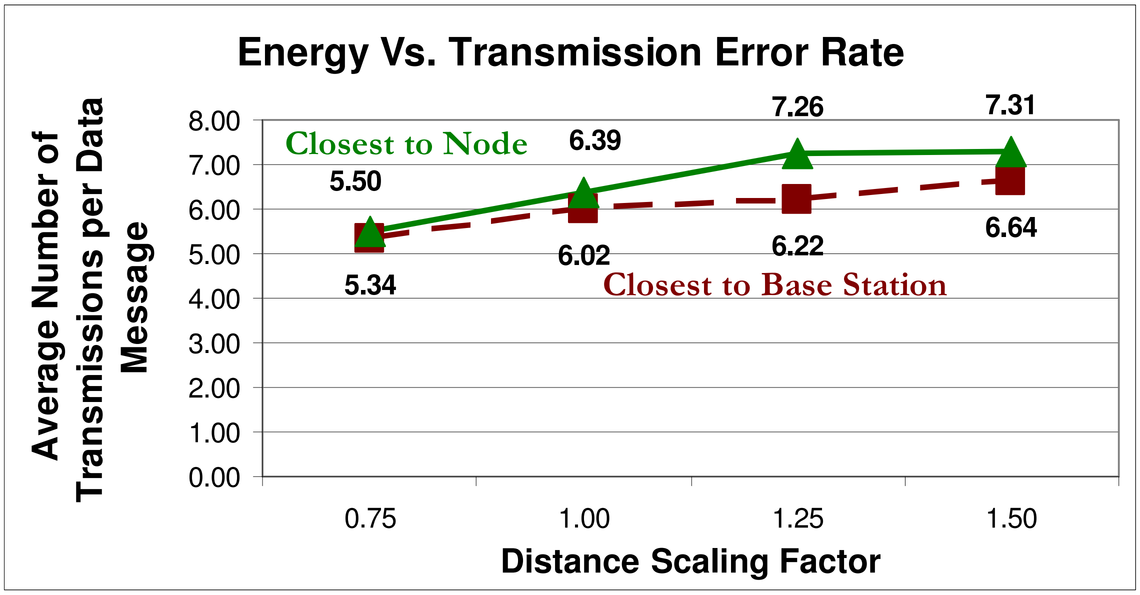

Figure 13: The average number of radio transmissions sent per data message generated as a measure of energy usage.

error rates. The fault tolerant scheme will eventually break down when the DSF exceeds 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 base station method until we look at the energy graph (Figure 13). This reveals that the Closest to base station method generally uses slightly less energy. Looking at the DSF of 0.75 case we see that the energy used, much like the success rate shown in Figure 12 is almost identical. However, when we look at a higher DSF such as 1.25 we can see that the closest to base scheme is only sending 6.22 messages per piece of data while the closest to node is sending 7.26 which is a 14.4% increase.

While these results are promising the fault tolerant scheme is providing much more than the higher global success rate shown in Figure 12. Another, probably more important, benefit it provides is that its success rate holds fairly constant throughout the

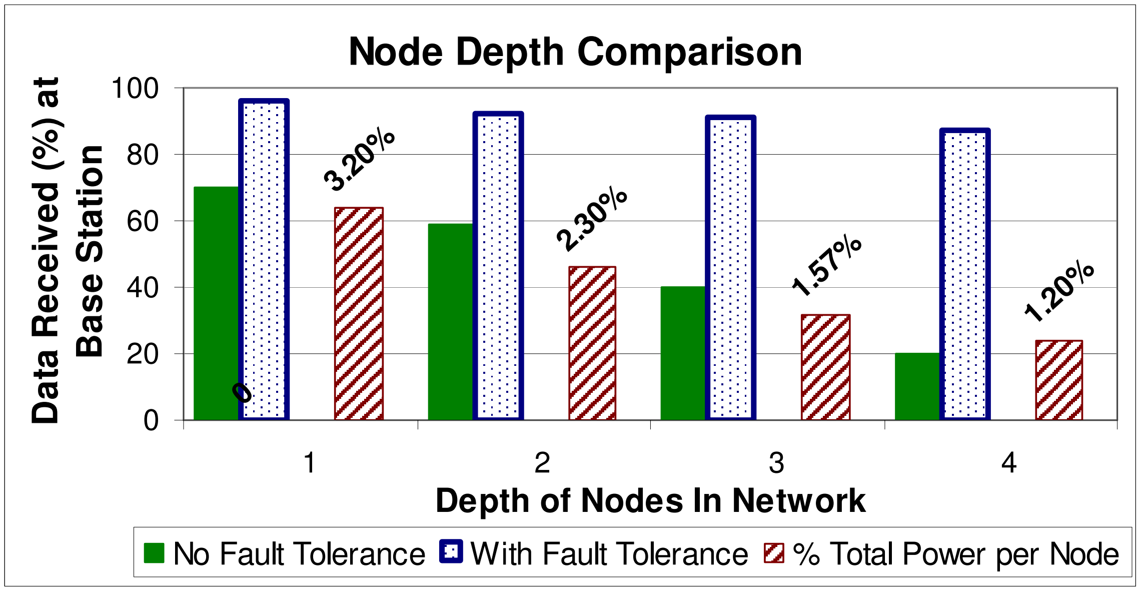

Figure 14: The success rates as a function of the nodes depth in the network. The dashed columns represent the percentage of the overall network energy that a node at this depth uses.

network. For example, the total message success rate in Figure 12 at a DSF of 0.75 without fault tolerance is 62.5%. However, this is not a constant success rate for all nodes in the network. In Figure 14 we show the success rate for each of the different node depths (hops to the base station) in the network. We see that at a depth of one 70% of messages succeed while at depth four it drops to as low as 20%. This clearly shows that while the total success rate is not so bad, the base station actually barely knows anything about the parts of the network that are further from it. We can see that with fault tolerance this problem is avoided and we have a fairly constant success rate for all depths. The benefit of this goes beyond just the base station having a good view of the network at four hops because there will be applications where far more than fifty nodes are required. Without re-routing there is almost no point to trying to expand the network as the success rate will be close to zero on any nodes further out than four hops. With re-routing however, we see only a slow gradual decay of the success rate as the network expands. Figure 14 also confirms that the further out from the base station the less energy a node requires. We can see that nodes of depth one use twice as much energy as nodes of depth three. In order to deal with this the density of nodes in a mote deployment should increase as they get closer to the base station, or alternatively the nodes closer to the base station could be outfitted with a larger energy supply.

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

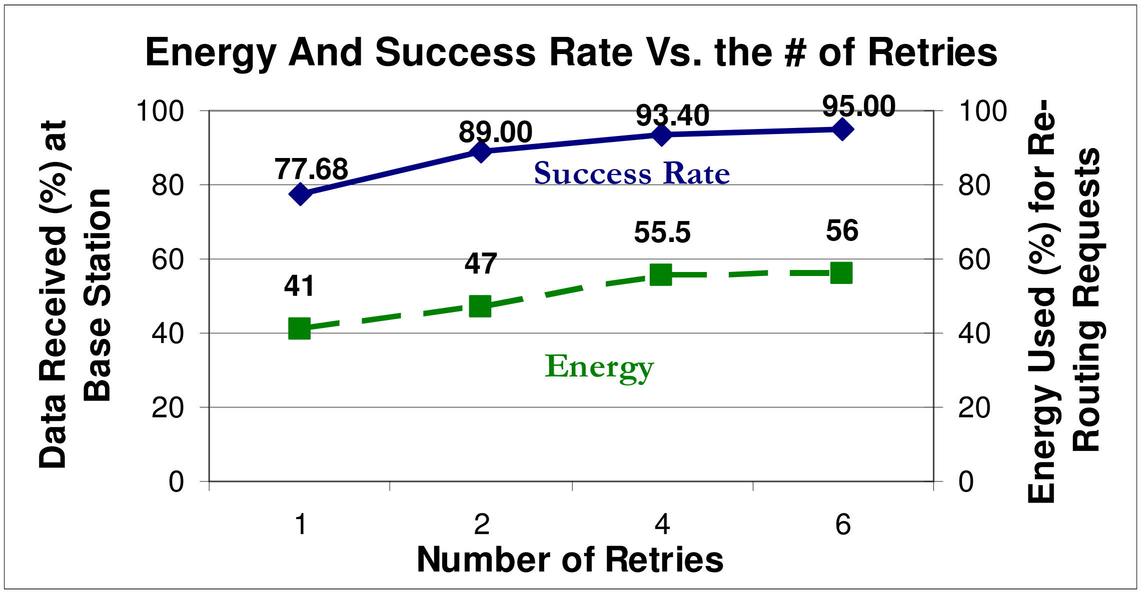

Figure 15: The effect of the retry threshold (left axis, solid line) on the data success rate and the percentage of energy used by the system re-routing algorithm (right axis, dashed line).

the effect on the success rate of data reaching the base station and the energy used by the algorithm. The results are shown in Figure 15. Examining the graph we see that while there is a notable 12% difference between one and two retry attempts, adding more retries gives diminishing returns of 4.4 % from 2 to 4 retries and 1.6% from 4 to 6 retries. Most applications would likely decide to go with the two retry attempts but if each and every packet of data was of the utmost importance they might go as far as to use four or even six for the maximum number of retries. Another possibility would be a system that uses a different number of retries for messages that are more important.

In Figure 15 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 two, more of the energy is being spent on re-routing messages but the overall success rate is not improving by much. Second, this shows that even in a harsh environment, using two 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 until we look at a simple approximation for the energy overhead of a multi-path alternative. Assume that messages fail on their way to the base station with the same rate of 60% and that they do so half way from the source node to the base station such that a node of depth *h* has a 60% chance of causing a resend of *h/2* messages. On successful messages we need *2h* messages to get the data to the base station and the ACK packet back to the source. This means that even if every message is successful the energy contribution from the ACK packets is 50%. This number is already higher than our overhead with a 60% failure rate and when compared to our algorithms zero overhead on successful messages looks even worse. In the case of failures *2.6h* (*2h* for a successful message plus *.6h* caused by the 60% failure rate) messages are needed. So, on average each message requires *2.3h* messages and the power overhead is 57%. This number would be much higher if we had taken into account failures on ACK packets, multiple failures for the same message, failures that occur closer to the base station requiring more overhead and the fact that nodes with larger *h* have a higher chance of failing.

CHAPTER 4

NEIGHBOR SELECTION SCHEMES WITH LEARNING

4.1 Motivation

The neighbor selection schemes that we have examined so far are all relatively static in nature. It is possible that the multi-hop router decides to change the assigned parent of a node (and conversely its depth in the network) but this is a rare event which becomes even rarer as the network routing tree settles into a steady state. While these algorithms are able to increase the reliability of the network substantially, the fact that they are static remains a serious flaw. The main conceptual reason for this is that if two nodes have the same depth, this does not mean they are necessarily geographically close.

Consider the example, illustrated in Figure 16, where some node *A* wishes to send a message and has two neighbors, *B* and *C*, which have the same depth. The static algorithms would treat these nodes the same. However, it could be that node *B* is geographically close to node *A* and a good choice to route through while node *C* is far away, just barely in radio range, and hence has likely a less reliable radio link than node *B*. While it might be that node C is far enough away that node A almost never hears it

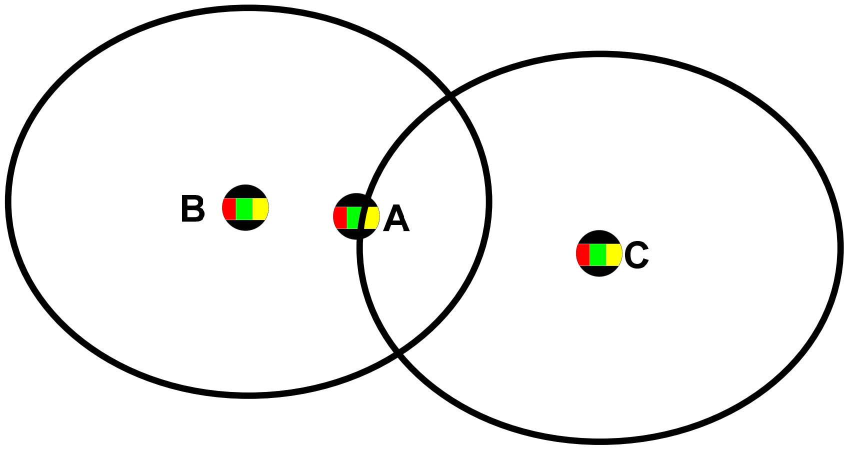

Figure 16: An example of a problem that can occur when nodes make decisions simply based on depth in the routing tree. The circles represent the radio range of node B and node C.

and hence does not have it on the neighbor list very often, problems can arise if node *A* only hears one out of every ten messages from node *C* but has not removed it from the neighbor list. We can not solve this problem by simply removing nodes from the neighbor list if we do not hear a lot of traffic from them because the node may be a very reliable link and just not have dad any data to send due to a lack of events in its sensor range. This means that even though node *A* does not hear from *C* very often, *A* does not know if this is because *C* is on a poor radio link or just does not have much to say. We wish to try and lessen the negative effect that this has on our network despite the lack of geographical information from the nodes. In order to do this we will make the nodes learn and modify their behavior based on their past successes and failures with each neighboring node.

4.2 Approach

The general process of the learning scheme will be the same as was shown in Figure 8 and discussed previously. The only difference will come in the step when we run the next best neighbor selection scheme, this step 5 in Figure 8. The new scheme will create a total score for each of the neighbors based on two factors. The first factor which we call the "static" factor is generated using one of the previous algorithms, for example, closest to neighbor. Each node depth is given a point value under this scheme, for example a node may choose to give 10 points to nodes one hop closer to the base station, 5 points to nodes two hops closer to the base station and 2 points to nodes three hops closer to the base station.

The second factor that contributes to the total score is the "learned" factor. The learned factor is kept track of independently for each of the nodes neighbors. The learned factor starts at some initial value and then is modified whenever a node asks another node to forward a data packet for it. If the neighbor does successfully forward the packet its score will grow larger by some value and if it fails to forward its score will decrease by some value. These two values do not necessarily have to be the same; nodes could add one point to a neighbor that properly forwards a packet but subtract two points whenever a neighbor fails to forward the packet.

Another important aspect is to balance the maximum possible score from each of the two factors as well as their respective starting points. We would like the numbers to work out such that for the first few packets sent by a node it places most of its emphasis on the base score and only once it has learned a bit about its neighbors will it start to favor the learned score. Once the node has sent a substantial number of packets the base score should have little affect on the neighbor selection process.

Once the node has calculated the total score for each neighbor it has to decide which neighbor to send to. There are two different ways to do this. The simplest way is to examine all of the scores and find the neighbor with the highest possible score. This neighbor is then picked and asked to forward the message. We call this method deterministic because given a certain set of node scores it will always select the same node. A slightly more complicated method is to use the scores as weights in a random selection process, we call this method non-deterministic because the node with the highest score is not necessarily selected during a given transmission, it simply has the highest probability of being selected for a given transmission.

An example of these two behaviors is a node with three neighbors, two with score 15 and one with score 20. In the deterministic version of the algorithm the neighbor with score 20 will be chosen as the next best neighbor until it fails to the point that its score drops below 15. In the non-deterministic case for this same example the sending node will add up the scores, in this case 50, and generate a random number from zero to this sum. The sender then will select the node that the number corresponds to. In our example if the random number is between zero and 15 the first neighbor will be selected, if it is between 15 and 30 the second neighbor will be selected and if it is between 30 and 50 the third neighbor would be selected. This method is non-deterministic because it will not always ask the same node to forward when presented with a certain set of scores.

We expect that the non-deterministic method will perform better than the deterministic method in situations where long transient (losing radio contact for as long as a few minutes) and permanent failures occur more often than errors caused by the wireless environment. Consider the case of a neighbor that has been performing well and forwarding all messages until it suddenly undergoes a transient failure, the deterministic method will continue to try sending through this node, failing each time, until its learned score finally decrements past some other node. In this same situation the non-deterministic approach may select a different node and succeed on each attempt. Conversely we expect the deterministic method to work better if errors are predominantly caused by radio transmission problems. This is because if a node sends to a neighbor with the highest score and the error is caused by a packet collision or corrupted data bit the next transmission has a high probability of being successful, the deterministic method will likely try this node again while the non-deterministic method could end up trying a lower probability neighbor.

4.3 Results

We first compare the success rate and energy that is used with and without learning. We would hope that with the learning algorithm added we would have an improvement in both success rate and energy usage. This stems from the fact that if nodes know who the better choices are to forward to, they should save energy because less retries are required and likewise the success rate should go up as we are only trying the “better” nodes. This comparison is shown in Figure 17. Here we see both the energy and success numbers for the cases of with and without learning. We can see that with learning we have either equal (at very low environment error rates) or better throughput to the base station with a 5% difference when the DSF is 1.50, this can be attributed to nodes on the outskirts of radio range receiving low scores and hence not being tried. Additionally, the energy used with learning is either equal (again at very low environmental error rates) or better in all cases due to having a higher probability of succeeding on the first guess.

We next examine the differences between the deterministic and non-deterministic approaches. As mentioned we expect that which one performs better would depend on how prevalent transient node failures are. To this end, we examine the effect of increasing the rate of transient failures while holding the radio environment constant. Transient failures are important because they model a different failure mode than a bit level transmission failure. When a message fails due to a bit error or a message collision the node that was the intended recipient will still be available to receive the retransmission whereas if the node is in a transient failure state it will not be able to.

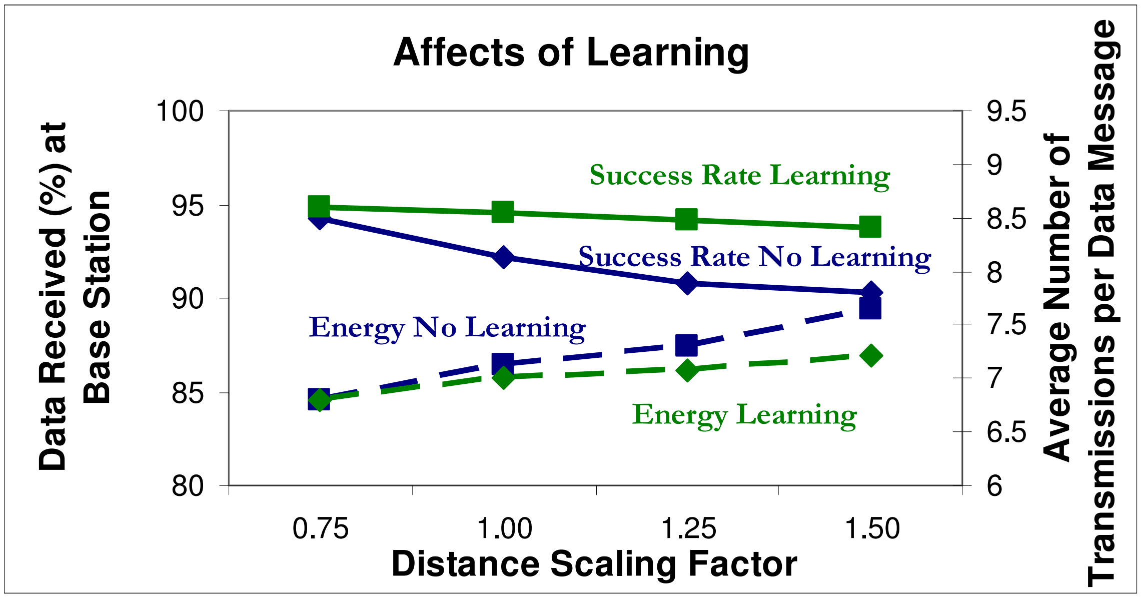

Figure 17: Comparing the success rate and energy between using learning and not using learning. The solid lines represent success rates and align with the left hand axis, the dashed lines represent energy and align with the right axis.

In Figure 18 we see compare the learning algorithms with the two different methodologies. The simulation runs involved fifty nodes and used the DSF of 1.50 in modeling the radio transmission error rate. We see an interesting behavior that when errors due to transients are low (on average 6 nodes with transient faults at any given time) and the radio errors dominate, the deterministic algorithm works better with a success rate of 75% as opposed to 70% in the non-deterministic case. At as the rate of transient failures increases to about 12 nodes with transient faults, we now have a 5% advantage in favor of the nondeterministic algorithm. As we expected, the deterministic method is better at figuring out what nodes have good quality radio links and sticking with those nodes whereas the non-deterministic method is more robust in the face of nodes undergoing longer errors that will affect subsequent message transmissions.

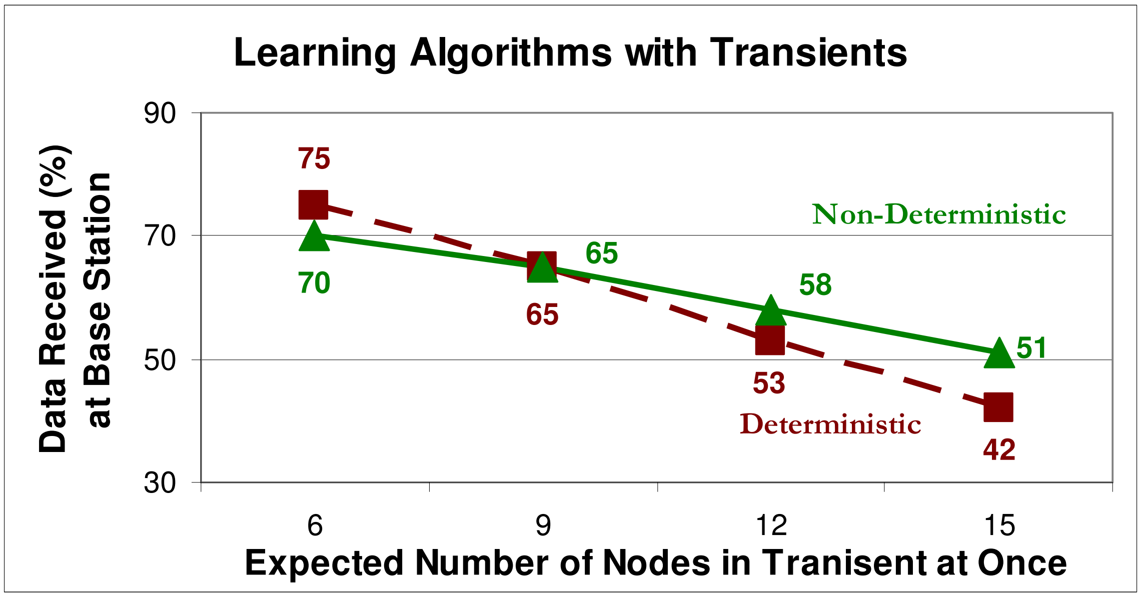

Figure 18: The success rates of the two different learning schemes as the rate of transient failures in the network varies.

Due to the nature of the learning algorithm there are some small tweaks we can try. Consider that there are a number of different rates which we can control. The results shown in Figure 18 used a linear increase and decrease (one positive point on a success and one negative point on a failure) for the learned score. One of the problems with this is that the nodes learn very slowly. In analyzing the simulation output it was clear that there were a number of instances where the learning was so slow that by the time it learned that a node was undergoing a failure, that node was already recovering from the failure. This would sometimes lead to a case where the algorithm would actually be performing worse than with no learning at all.

In order to avoid these types of problems we want the algorithm to learn faster. In Figure 19 we show that just one small tweak to the learning can increase the success rate by about five percent. Instead of using a linear increase and decrease we reward consecutive failures and successes, i.e., the first time a node fails to forward it loses one point, if it then fails again it loses 2 points, then 3,4,5, etc. The gain in success rate would be even larger in a higher throughput network. This is because even with the faster learning if a node only tries to send to a failed node 2 or 3 times while it is in a failed state there is not enough time to truly react to the situation. Most of the benefit in our experiments came in the nodes that receive a lot of intermediate routing traffic, and hence in a busier network more nodes would be seeing these benefits.

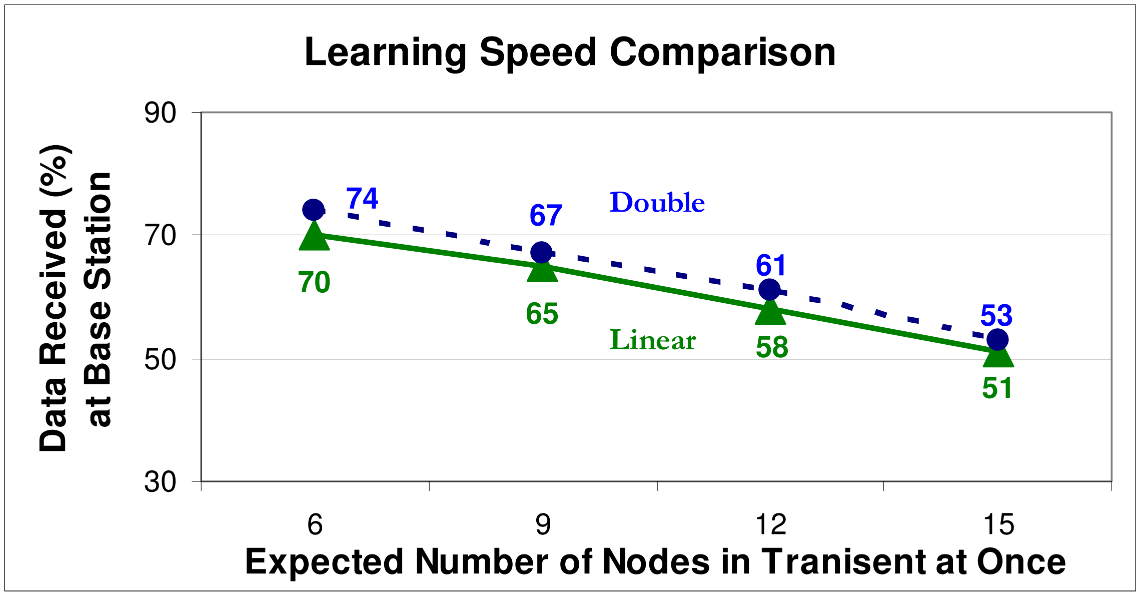

Figure 19: The benefit that a small change in the learning mechanics can provide. The solid line represents a linear increase or decrease in score whereas the dashed line rewards nodes by doubling the score increase or decrease whenever consecutive successes or failures occur.

CHAPTER 5

COORELATED EVENTS

5.1 Motivation

In all of the experiments that have been discussed so far events occurred at a single node independently of other nodes, so long as the node was not in a transient error state. This was done primarily because the simulator does not have any built in support for correlated events. However, in a real world situation it is likely that events will be tightly correlated as opposed to independent. Most sensor network applications involve nodes with a sensor looking for some type of physical event, be it sensing lightning strikes or a battlefield situation where nodes are detecting movement through a field. Most of these types of events are likely going to be large enough that multiple nodes in the same general area will see it at the same time. We would like to be able to model these types of events into our simulations to see what sort of effects it has on our system and what changes it might imply.

The biggest problem that arises when correlated events exist is that a certain area of the network will be flooded with many different messages which will cause forwarding queues to fill up on nodes and also increase the probability of radio collisions in the area of the event. This may have a negative impact on both the success rate of messages reaching the base station as well as the energy being used. This means that fault tolerance will likely play an even bigger role than it did in previous results in getting information to the base station. We expect that once the data makes it through the first "burst" of many nodes reporting at once that the throughput should remain fairly close to what it was when we had events appearing independently from different areas of the network. The question is how good we can do during this initial burst.

5.2 Approach

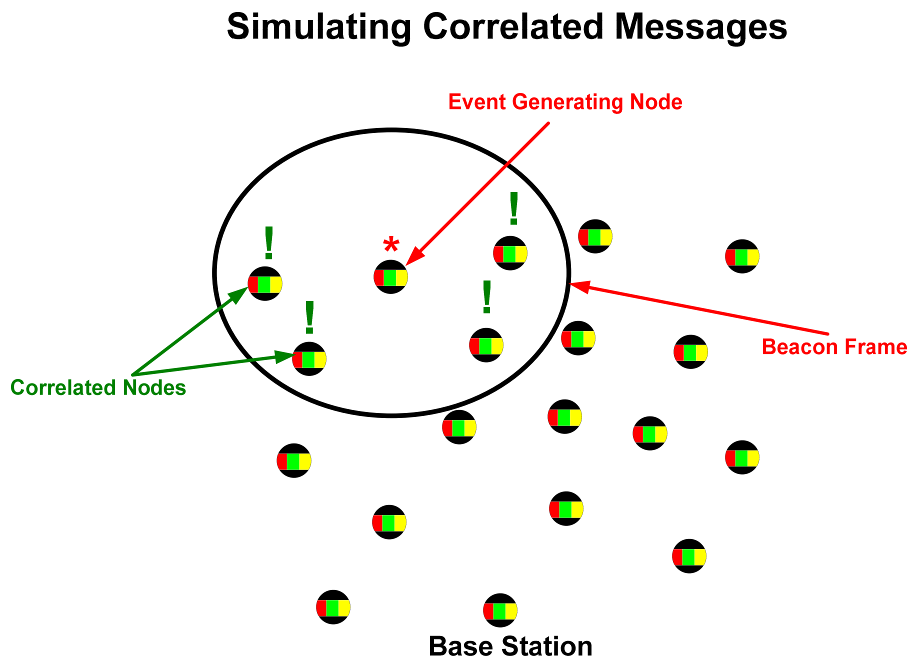

Figure 20: The process for simulating correlated events. The starred node is the node that generates the event, the nodes with exclamation points check to see if they will generate a correlated event.

TOSSIM does not have any built in support for correlated events so we will simulate them in the software that runs on our nodes. We create correlated events by still be generating events at random at each node as they were before but simply at a lower rate. Once an event is generated at a node that node sends out a special packet with a particular flag bit set high, we call this the beacon frame. When the neighbors of this node see the message with the flag bit set high they know that the packet is an original (just generated) event and that they should perform a probabilistic calculation to see if they will generate a correlated event. We call this probability the correlation factor as increasing it will result in more nodes participating in each event. It is important to note that only this first message immediately following the event generation will have the flag bit set high. This restricts the possible correlated nodes to those within radio distance of the generator and in practical terms creates a correlated event which has the original generating node at the center of all the nodes that report the event. This process is shown in Figure 20.

5.3 Results

To test whether having correlates messages has a large impact on the throughput of the network we performed a number of experiments using the same fifty node layout, held the number of retries constant at four and did not alter the decision making process or leaning behavior in any way. There are two variables here, the first is the distance scaling factor and the second is the amount of correlation between nodes.

We performed tests using the usual DSF values of 0.75, 1.00, 1.25 and 1.50 with different correlation factors. By correlation factor we refer to the probability that a node generates a correlated event along with the neighbor that originally generated the event. If the correlation factor is 10% then we can expect 10% of a node's neighbors to also generate a message. The three correlation factors that we examine here are 15%, 30% and 60%, these we chosen because they show the break point where correlated messages really start to influence success. To give a better idea of what these percentages represent, in the case of a CF of 15% we saw on average of 3.28 nodes seeing each event. With a CF of 30% we see 5.19 nodes seeing each event and with a CF of 60% we see 8.33 nodes for each event. Note that the number of nodes reporting an event does not quite double when the correlation factor is doubled; this is due to the original node always reporting the event. In the case of CF being 15% we actually have 2.18

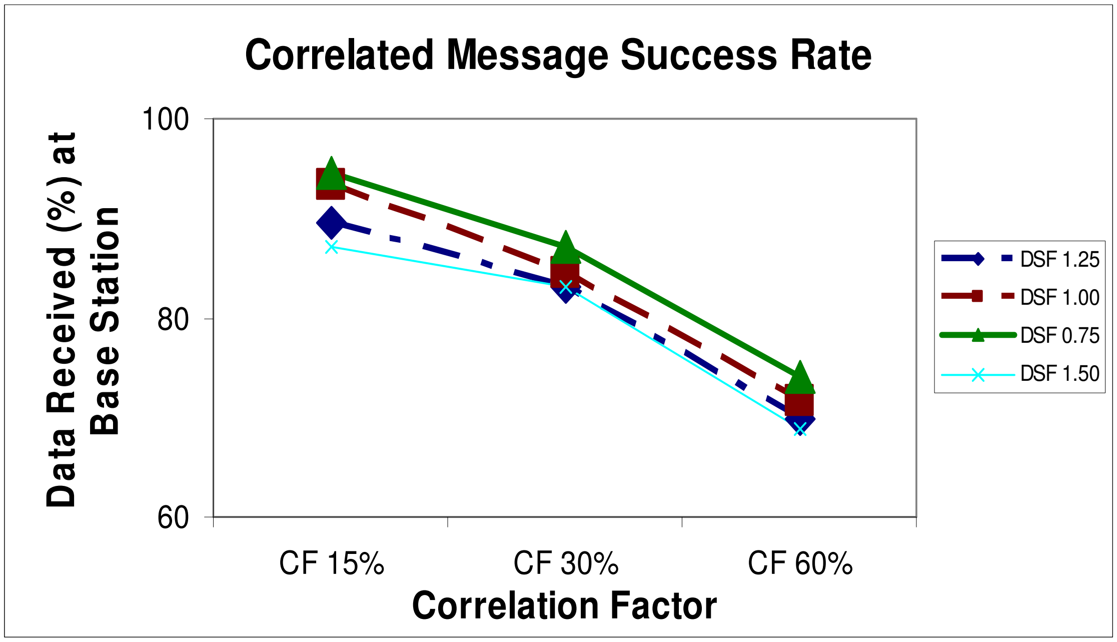

Figure 21: The effects of the correlation factor and the distance scaling factor on message success rates.

"correlated" nodes which also see the event, doubling this and adding the original node we get a value of 5.36, very close to what we see with the CF being doubled to 30%.

In Figure 21 we have plotted for the three different correlation factors the percentage of data received at the base station. The different data plots represent the four different values used for the distance scaling factor. This graph shows us some very interesting things about the affects of correlation. We first examine what happens when we move from a 15% CF to one of 30%. In this case it is clear that the correlation factor affects the throughput to the base station but it is not the dominant factor. This can be seen by looking at the success rate at 15% CF and DSF of 0.75. At this point we see a 93% success rate. From this point, if the DSF is increased to the worst possible scenario of 1.50 the success rate is reduced to 87%. Likewise, if we keep the DSF at 0.75 and instead increase the CF to 30% we also end up at an 87% success rate. This means that even a small increase in the CF is equivalent to a large increase in the radio error rate.

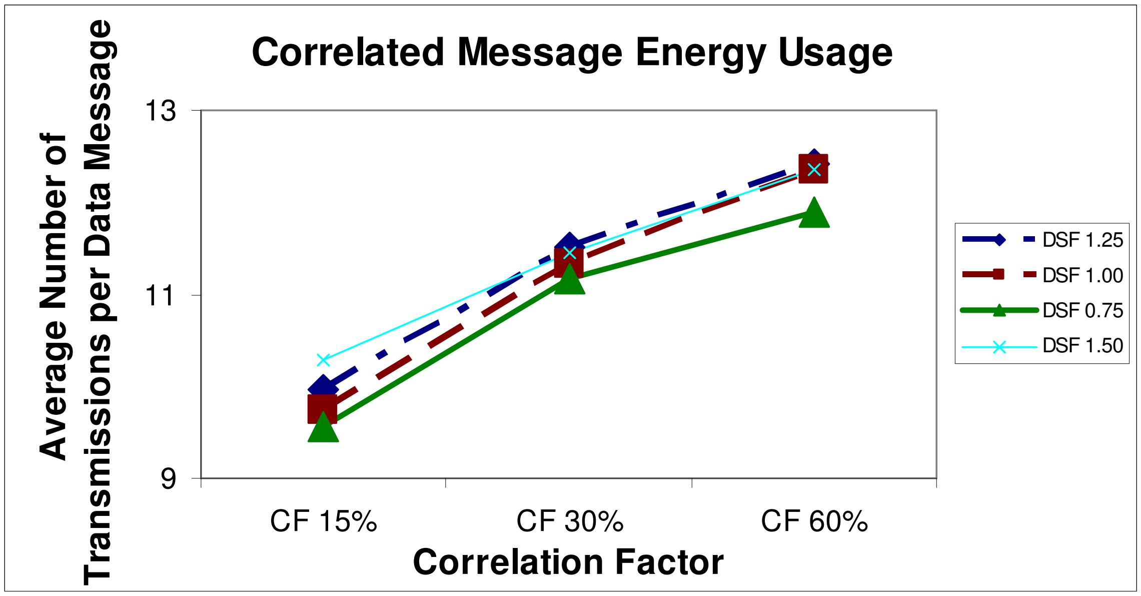

Figure 22: The effect of the correlation factor and the distance scaling factor on the number of messages required for each event.

Next we look at what happens if the CF is again doubled from 30% to 60%. If we look at the point where CF is 30% and the DSF is 0.75 we are at the 87% success rate mentioned previously. If we now increase the DSF to 1.50 the success rate sees a drop similar to in the previous case, down to 83%. However, if we keep the DSF at 0.75 and instead increase the CF to 60% the success rate drops to 74%. At this point the correlation factor has reached the point where it influences the simulation results even more than the radio environment in terms of throughput.

We now examine Figure 22 which looks at the energy consequences of correlated messages. While the increase in collisions is a strong force in the success rate of messages it is an even larger factor in the number of messages that nodes have to send and hence in their energy usage. If we look at the point where CF is 15% and the DSF if 0.75 we are seeing approximately 9.04 messages per event over its lifetime in the system. Increasing the DSF all the way to 1.50 only increases this to 9.78, however if we increase the CF to 30% the number of messages per event shoots up to 10.67, more than double the increase from doubling the DSF. This point is shown further by the general behavior of the curves as we increase the correlation factor. We can see that at CF of 15% the curves are still distinct but as we move up towards a CF of 60% the number of messages is almost identical regardless of the DSF.

5.4 Combining Correlated Messages

In section 5.2 we saw results showing that the more nodes which report an event the harder it is for each individual message to make it through successfully. It was stressed that the main motivating factor is that the area of the network which contains the event will have a large burst of traffic that can increase collisions and hence the failure rate of data making its way through the system. One interesting extension of this is to look into ways to decrease the number of messages that we are sending by combining similar messages into a single message. We have implemented a basic system in order to try and quantify the difference in both success rate and energy usage when there is a smaller number of messages.

The main idea of the combination approach was based around assigning different weights to messages that are now going to represent multiple other messages. When a node receives a message that it is supposed to forward along through the network it will no longer immediately send the message along but instead hold it for a short period of time. During this frame other messages that come in are checked to see whether they have a similar payload and if they do, they will be combined into one message with a "weight" field set to the sum of the individual weights.

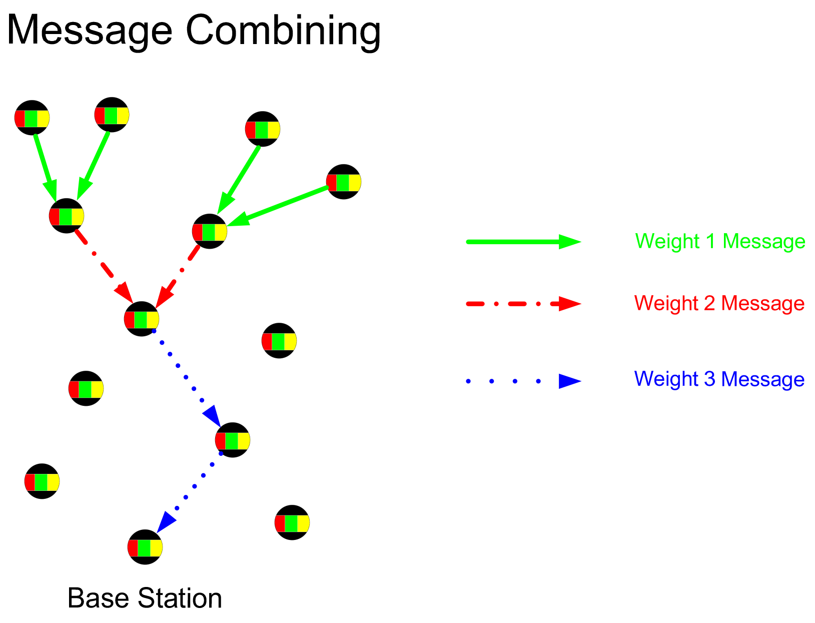

Figure 23: An example of how messages are combined in order to lower the number of transmitted messages.

This time frame is a parameter which when set to be too short could cause potential combinations to be missed but when set to be too long could have a detrimental affect on the latency of the network. This parameter would have to be adjusted based on latency requirements of a particular application. It is also important to make sure that this timer is not set longer than the time at which the fault tolerant software assumes that a message failed, otherwise the fault tolerant software would always think that its message was not received when it actuality it was received but it being held to check for combinations. This would quickly escalate into a situation were nodes were not only failing to have their messages heard at the base station but were also quickly draining their energy supply.

Another interesting point is how do we make the decision that two messages are "similar" and can safely be combined. We assume that there is some byte(s) that specify the sensed value which represents the event. Since we can not expect the sensed vaules by two nodes to be identical we also select a precision factor. For example if the precision factor was set to *25* and the first message received had a value *850* then any message received (during the waiting period) with a value between *825* and *875* would be combined into a new message with the value corresponding to the average of the values before the messages were combined.

While this system is fairly simple it still goes a long way in accomplishing the goal of reducing the number of messages generated by the event. Since all of the nodes who sensed the correlated event are in the same geographic area there is a high probability that they forward through the same intermediate nodes. If instead of having to copy four different messages with the same event through five hops to the base station we only have to move four copies through one or two hops and then have one message with a higher weight for the last three or four hops we can clearly save a great deal of energy and possibly increase the success rate. An example of message combining is shown in Figure 23 where there are four nodes that initially see some event and send a message with weight one. The two nodes that route these four messages each combine two messages with weight one into a single message with weight two and send this message on to their parent node. This parent node receives two messages of weight two and combines them into a new message of weight four, this message is then forwarded through the network until it reaches the base station. One potential problem is that if we combine four messages from the same event into one new message and then that one new message is lost on its way to the base station, the penalty is much higher than if we had lost one or even three of the original four messages.

5.4.1 Results

We performed simulations to examine the effects of message combination on the energy usage of nodes as well as the percentage of original messages that successfully make it to the base station. In these experiments we held the layout constant and used fifty nodes. We also set the number of retries to four and did not vary the selection or learning protocols. The two parameters that are varied are the distance scaling factor and the correlation factor, similar to the results shown in section 5.3.

In Figure 24 we look at the affect of message combinations on the success rate of messages in the system. As in previous experiments we use all four distance scaling factors though we will only look at two different correlation factors, that of 30% and 60%. We will look at each of these correlation factors without combing similar messages and then with combining enabled so that we can see the comparison, this makes up the

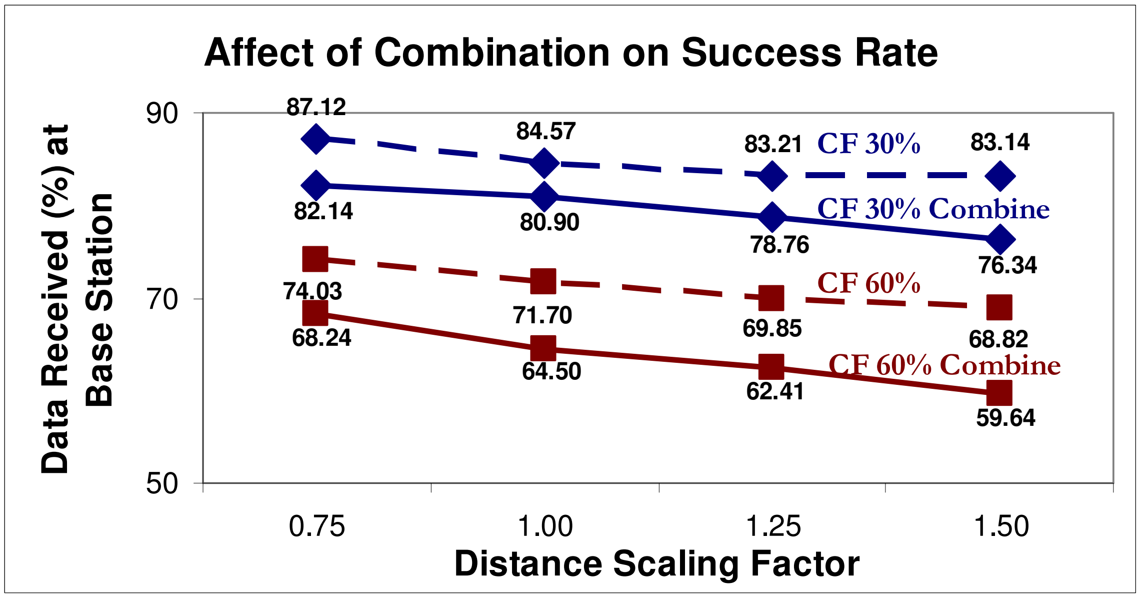

Figure 24: The effects of the combining correlated messages on the success rate of messages in the network. The dashed lines represent the case of not combining and the solid lines represent what happens when we start combining.

four different plots. Looking at the graph we see that the fear of lowering the systems success rate is indeed a real issue. The difference does not greatly depend on the DSF or the CF as the difference between the two stays relatively close. At the two extremes we see a 5% drop (from 87.12% to 82.14%) with a CF of 30% at DSF of 0.75 and a drop of 7.5% (from 68.82% to 59.64%) with a CF of 60% and DSF of 1.50. The reason behind this drop is that we are combining messages into a new one that now carries more importance. This means that if we lose a message of weight four we pay the penalty as if we had lost four messages of weight one.

One important note is that these numbers represent the probability of a single message reaching the base station. This means that while we are slightly decreasing the probability of a message reaching the base station, the probability of each event being heard at least once is still greater than 99% in all cases (due to the fact that all of the, on

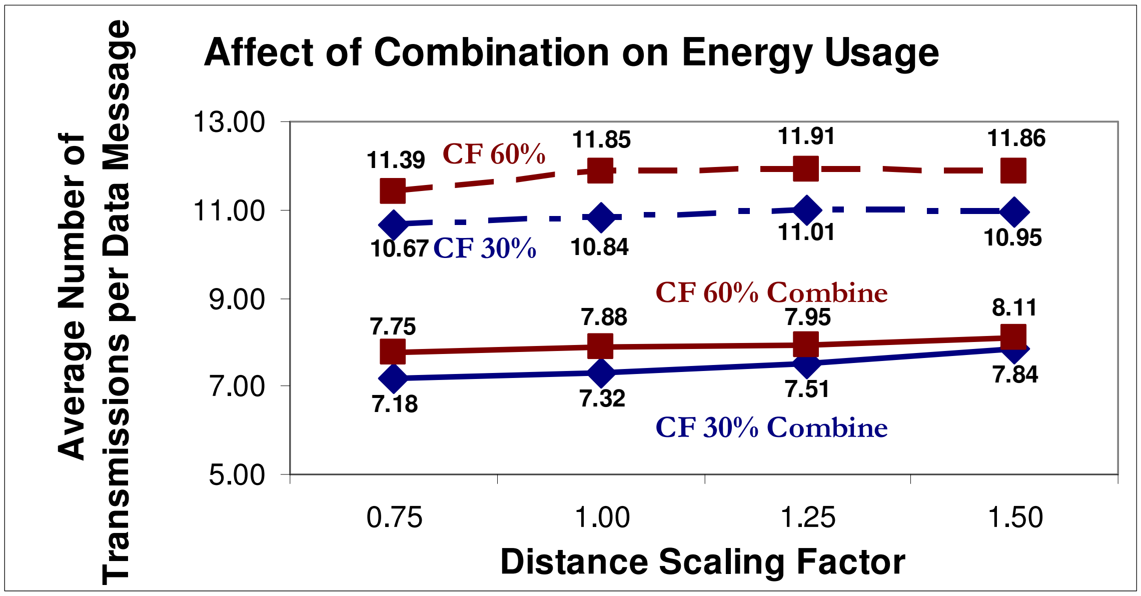

Figure 25: The effects of combining correlated messages on the number of transmissions required for each original message. The dashed lines represent no combinations and the solid represents the case of combining similar messages.

average, 5+ messages would have to fail for the event to get unnoticed). This means that we are not in any danger of missing an event entirely, but only in danger of underestimating the magnitude of the event.

While the loss of messages is certainly a concern we hope that it can be justified with a savings in the number of transmissions required. In Figure 25 we see that this should be the case for all but the most stringent applications. We plot the same four data sets as in Figure 24. Consider the data point with a DSF of 0.75 and CF 60%; in Figure 24 we saw that combining messages lowered the success rate by 5.5%, however we see in Figure 25 that the number of transmissions drops from 11.39 to 7.75, a 32% decrease! When we look at another point with a higher DSF of 1.50 and CF of 30% we still see an improvement from 10.95 to 7.84 or 29% savings as opposed to the 7% difference that we saw in success rates. This shows that our energy savings far outweigh the loss in message success rate, especially when we consider the previously mentioned fact that we are not missing events, just possibly underestimating their magnitude.

5.5 Weighted Retries

We have previously discussed the reason that correlated events are important and how to implement them in our simulations. We saw that when more nodes see the event, not only the amount of energy increases but the probability of a particular message making it to the base station goes down. We attempted to resolve some of these issues using the combination scheme discussed in 5.4 and saw great success in reducing the amount of energy used. As discussed in 5.4.1 however, the success rate of messages from their original source to the base station actually went down.

When first considering this result it seems counterintuitive. If we lower the amount of traffic flowing through the system, why should we see less of our messages at the base station? Shouldn't the lack of traffic lower the possibility of collisions and interference and increase the success rate? The problem here is that our fault tolerant scheme was built assuming that all of the messages deserved the same treatment. If the number of retries for a particular application was set to 2 then every message will receive 2 forwarding attempts from nodes along its route in the case of failure. However consider a message that has a weight of 5. This means that 5 different nodes reported the event and all of that information has been combined into one message. If we treat this message the same as a message with a weight of 1 then we are not appropriately scaling the level of protection with the importance of the message.

In order to try and fix this we looked into scaling the number of retries with the importance of a message. To do this we use a simple multiplier on the base number of retries that is provided for the application. Now, if the message has a weight of *k* the actual number of retries that are used for a message is *retries*k*. We expect that this would allow us to increase the success rate back up to (or perhaps higher than) where it was before we started combining messages. Additionally while scaling the number of retries will increase the energy usage somewhat it should still remain well below the energy that was used before we were combining messages.

5.5.1 Results

After implementing the weighted retries scheme the key points of interest were twofold. First, we hope that using weighted retries we can at least achieve the same success rate as before, and secondly, while we may sacrifice some energy we still would like to have energy savings over the initial case of not combining messages at all. In other words, we would like to see that by combining messages *and* scaling the number of retries appropriately we create a solution which is better on *all* accounts over just doing nothing. For these tests we again use a constant fifty node layout and do not modify the number of retries or the selection algorithm.

In Figure 26 we see results for all three of the scenarios that we have discussed; simply having correlated messages, combining the correlated messages when possible and finally combining correlated messages and scaling the number of retries based on the weight of the message. The first trend that we see is that the correlation factor's effect on the system is greater than anything else, this can be seen by the fact that even the smallest value with a 30% correlation factor is better than the best case with a 60% correlation factor. Additionally we can see that we have accomplished our first goal, when adding weighted retries we have achieved an equal or better success rate than we had before we

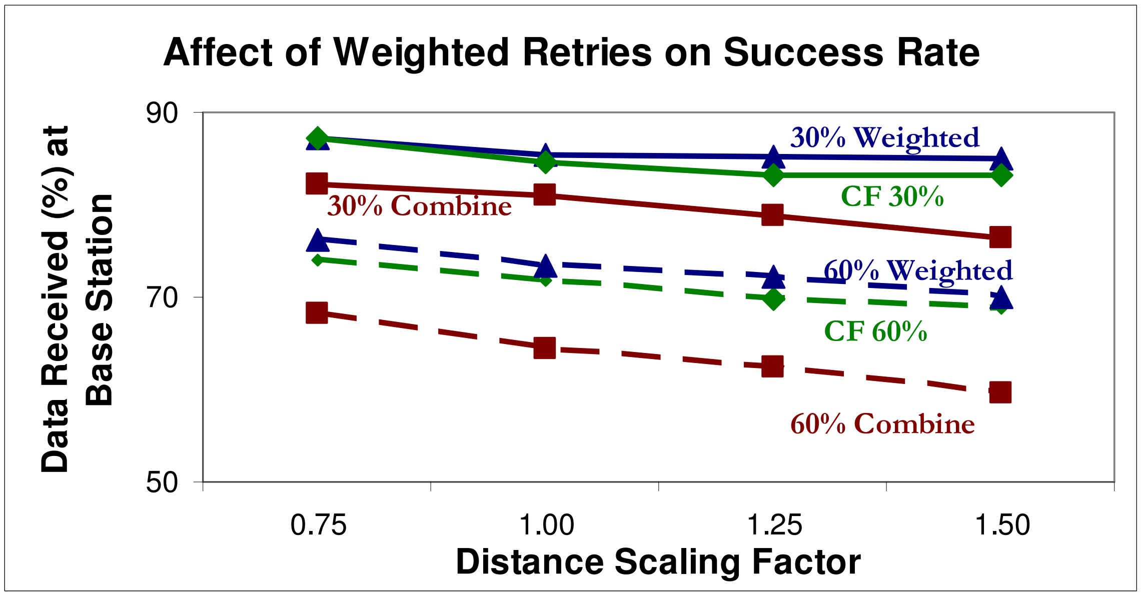

Figure 26: The effects of scaling the number of retries, based on the weight of a message, on the success of messages reaching the base station.

started combining messages. If we look at the case where the distance scaling factor is 1.50 and the CF is 30% we see that without doing anything the success rate was 83%. When we started to combine messages we saw a drop down to 76% but now that we have taken into account that some messages are more important than others our success rate is up to 87%. This same trend holds for every scenario that we tested.

We also need to see if we have met our second goal of maintaining much of the energy savings that we achieved when we started to combine messages. In Figure 27 which plots the number of transmissions necessary to receive the results in Figure 26 we see that using weighted retries uses more energy than when we did not use them but still saves a good deal of energy over the case without combinations. In order to quantify these savings consider again the point where the DSF is 1.50 and the CF is 30%. In the case of doing nothing the average number of transmissions is 10.95. When we combined messages this dropped to 7.84 and now that we assign weight to the number of retries, the

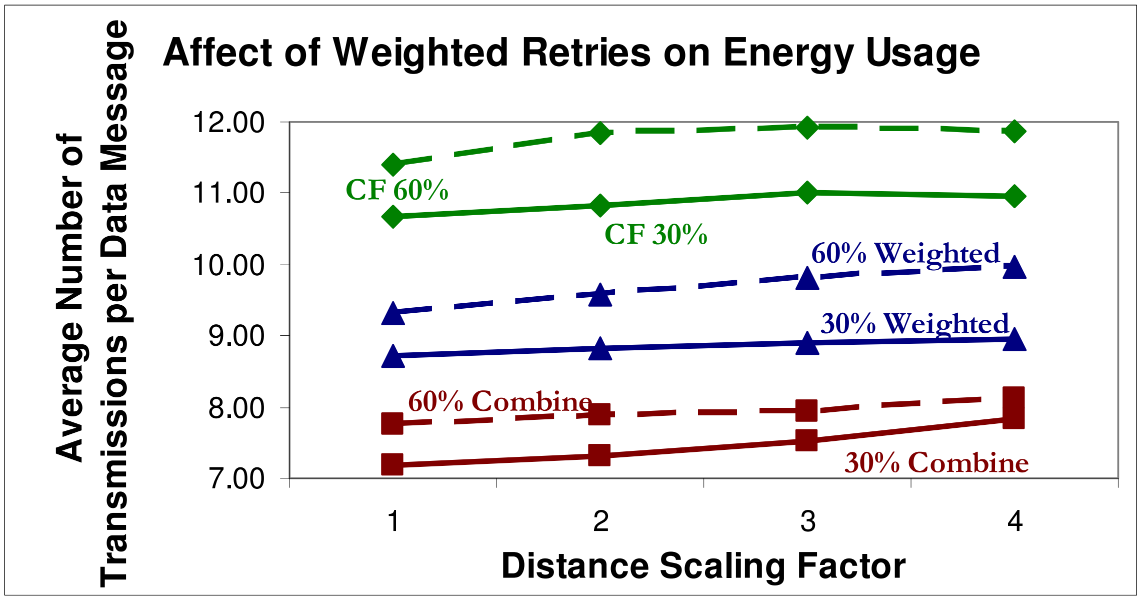

Figure 27: The effects of scaling the number of retries based on the weight of a message on the number of transmissions required for each original message.

number is 8.96. 8.96 is a 19% energy improvement over not combining and also has a 4% higher success rate. When we use combinations and weighted retries we improve both the energy and the success rate of the network.

One interesting point is that in some situations it may be considered worthwhile to not weight the retries and simply combine messages. This would make sense if energy was of the utmost importance and the success rates that remained after combining messages were considered to be good enough for that application. As mentioned previously even with the lower success rate from combining the messages but not scaling the retries more than 99% of all events are heard at least once, there is just a chance of not knowing exactly how many nodes heard the event.

CHAPTER 6

ADVANCED TOPICS

6.1 Duty Cycling

All of the previously discussed results had all of the nodes in the simulation run available at all times with the exception of nodes undergoing transient failures. No where did we have a way to turn nodes off to save energy in a controlled fashion (as opposed to transient failures which occur at random for indeterminate lengths of time) for a specified period of time. Duty cycling is an important technique in sensor networks because nodes are very sensitive to energy depletion. If nodes in the network start to fail at key locations then at best the throughput of the network drops and at worst the entire network can become disconnected.

In the sensor network literature duty cycling is generally handled at the MAC layer. When nodes are going to be in a low power mode most of the time, the biggest issue is making sure that the correct nodes are awake at the right time so that data can still be moved throughout the network. Duty cycling is useless if nodes are all randomly awake for 5% of the time resulting in a network throughput of almost zero. It makes sense then that the MAC layer is responsible for duty cycling as it is already tasked with keeping the nodes fairly well synchronized such that its specific MAC protocol can run properly.

The TinyOS default MAC layer, B-MAC, has a duty cycling mechanism built in that allows saving energy. The B-MAC implementation for the Mica2 motes in TinyOS is a part of what is called the CC1000 radio stack. The TinyOS distribution [1] includes a document discussing the radio stack and providing the table shown in Figure 28 describing the seven available duty cycles. It is evident from this table that duty cycling

Mode	Duty Cycle (%)	Max Packet Rate (pkts/sec)	Effective Data Rate (kbps)
0	100	42.93	12.364
1	35.5	19.69	5.671
2	11.5	8.64	2.488
3	7.53	6.03	1.737
4	5.61	4.64	1.336
5	2.22	1.94	0.559
6	1.00	0.89	0.258

Figure 28: The different duty cycling modes of the CC1000 radio stack for the Mica2 mote [1].

is essentially a bandwidth problem. When the nodes are turned on at all times in mode 0 they are able to send almost 43 packets per second but in mode 6 this number drops to less than one packet per second. We would like to see how these different modes affect the behavior of our fault tolerant software. Since all of the functionality has been implemented at the routing layer, we expect that it should not interfere with duty cycling until the data rate reaches the maximum that the mote can handle.

When performing simulation experiments using these low power modes however we ran into an issue with the TOSSIM simulator. While TOSSIM will allow the use of the CC1000 radio stack in simulations and allow function calls that modify the power mode, it will not actually emulate the changes in the MAC layer timing due to the change in power mode. This means that a work around is needed in order to test this duty cycling behavior in TOSSIM.

To do this we added another module just above where we send and receive all our radio transmissions to and from the radio stack. We then emulate the duty cycling behavior using timers such that when it is determined that the node would be in the "sleep" state it will not pass any data between the radio stack and the higher layers. We also provide a function that allows the higher layers to check if the node is currently in the sleep state or not. During the time that the radio stack is in the low power sleep mode the nodes can still detect new events and add them to the queue to be sent out when the radio stack wakes up. The bandwidth limitations that were discussed and shown in Figure 28 start to appear when the queues begin to overflow. This work around is an idealization in that the nodes are expected to be almost perfectly synchronized whereas in real world practice the MAC layer would certainly be imperfect. Additionally, in order to keep the simulation running at a reasonable rate, the size of the duty cycling period is larger here than it would be when implemented at the MAC layer, though the percentage of time of being asleep and awake remains the same.

The first set of tests which were performed involved a fairly low data rate. Events were generated at a rate of about 1.53 events per second in the fifty node network. This rate is slow enough so that it would likely never impose a problem if correlated events are not used. However, due to the correlated events, this equates to about 3.32 nodes that see each event in a small area, as these nodes try to send the messages through their

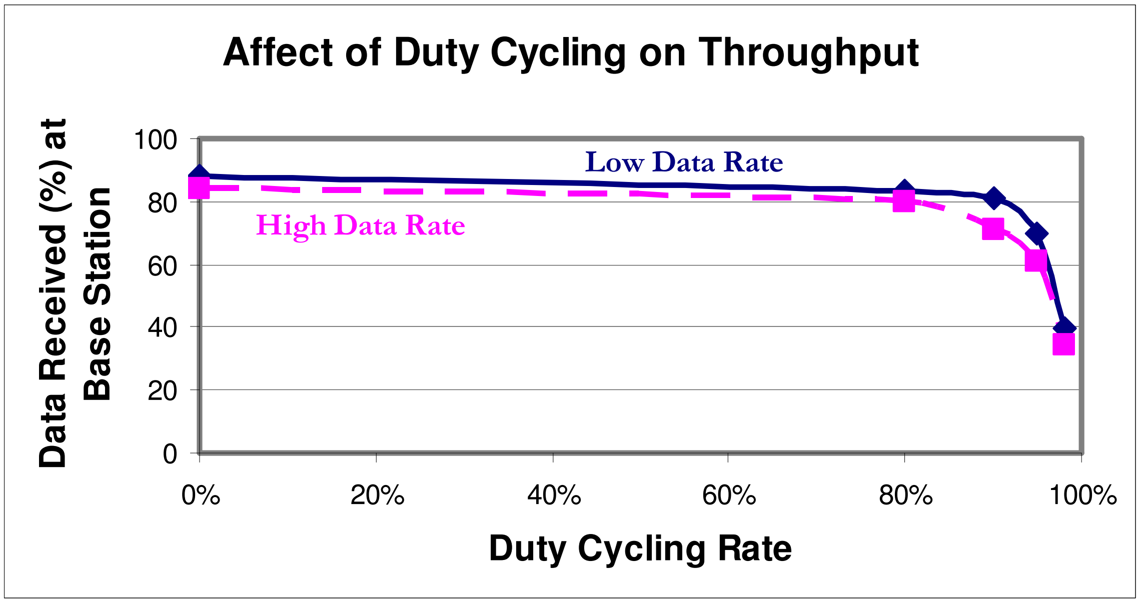

Figure 29: The effects of duty cycling on the throughput of the network.

respective parents, there could be anywhere from one to possibly ten messages in a small area of the network. We see in Figure 29 that this leads to the type of behavior that we were expecting, the throughput remains fairly constant until we reach the point where too much data needs to be sent for the current duty cycling rate, the queues begin to overflow in the node and the majority of messages are lost. This drop off occurred around the 95% to 98% duty cyle range corresponding to around mode 4 and 5 in Figure 28 which specify a rate of 2 and 6 packets per second for the two modes. This is right around the point where we expected our data rate to be too high for nodes around the correlated event. We then increase the rate at which events are generated by 50% and saw that the same general behavior occurred except this time the "knee" or failure point of the system occurred with the duty cycle 5% lower, this makes sense because with more messages to send the queues would start to overflow sooner.

In Figure 30 we see an interesting behavior that is caused by these throughputs.

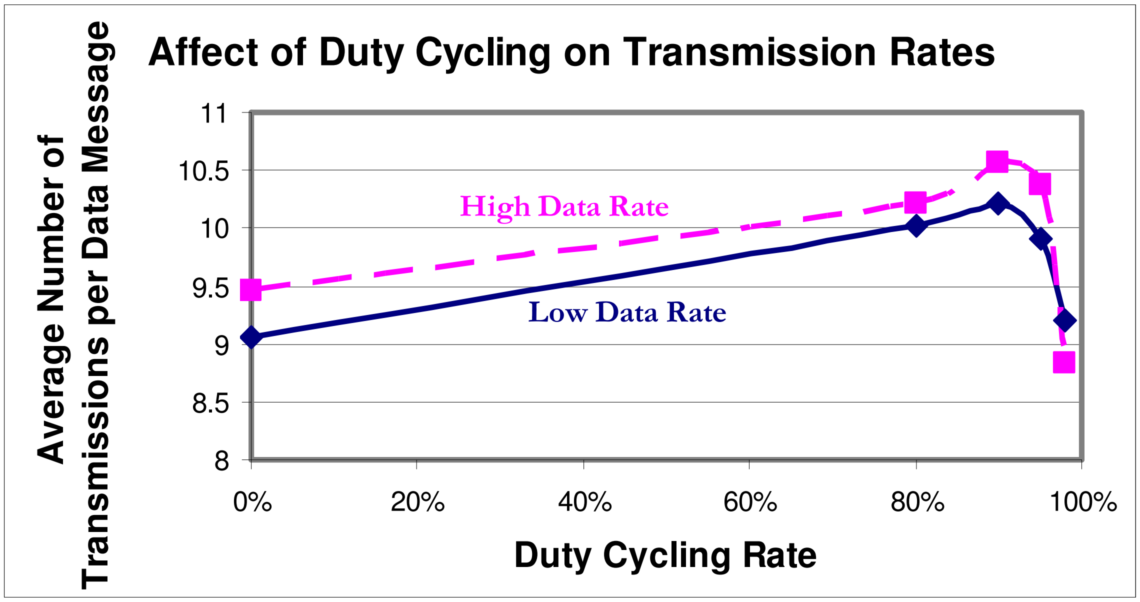

Figure 30: The effects of duty cycling on the number of messages being transmitted in the network.

As the duty cycle decreases, the average number of messages needed for each transmission goes up but when we reach the failure point it suddenly drops rapidly. This sudden drop is caused by the fact that most messages are not making it past their originating nodes and since no one has to forward the message through the network, the number of messages being sent drops suddenly.

In previous experiments we used the number of transmissions as a measure of energy as they were the only real variant between different sets of data. Now that duty cycling is employed, the energy equation becomes a bit more complicated. In order to estimate how much energy is being saved by duty cycling we need a rough approximation of how energy is used in a node. Using numbers for B-MAC on a Mica 2 mote [12] we make the following approximation. To transmit a packet we must send 36 bytes of information. It is given that each byte takes *416E-6s* and draws *20mA*, since the Mica 2 motes have a *3V* power supply we get the estimate of *898uJ* to transmit a packet. Similarly, the energy to receive a packet is estimated to be *673uJ*; less current is drawn while receiving data resulting in less energy spent when for receiving a packet. It is also mentioned that to sample the radio it takes *17.3uJ* and the default sampling rate is 10 times per second for a total of *173uJ*.

From these numbers we calculate the energy used per node per minute by looking at the total number of messages that were sent and received through the lifetime of the simulation. These are multiplied by their respective energy numbers and then added the total energy that is used just by being on and sampling the radio. This value is calculated by taking the *173uJ* per second and multiplying it by the number of seconds that the simulation ran for and then multiplying by the percentage of time that the node was awake. For example, in a one hour simulation with an 80% duty cycle, the energy from being awake is calculated as *.2*3600*173uJ*. In order to put this in units per node per minute we divide by the number of nodes and the number of minutes that the simulation ran for.

In Figure 31 the energy estimation is shown for the same experiments that were used to generate Figures 29 and 30. At a 90% duty cycle and a low data rate a node is using 2702uJ per minute as opposed to 13392uJ per minute at a 0% duty cycle. This is approximately one fifth of the energy and at this point the success rate of messages in Figure 29 is still stable. With the high data rate we are able to see that just before the point where data stops reaching the base station the energy actually spiked upward. This is because at this point the fault tolerant software is sending a lot of messages to try and get messages through. Just past this point the energy drops back off as most messages are not making it past their first hop in the routing tree

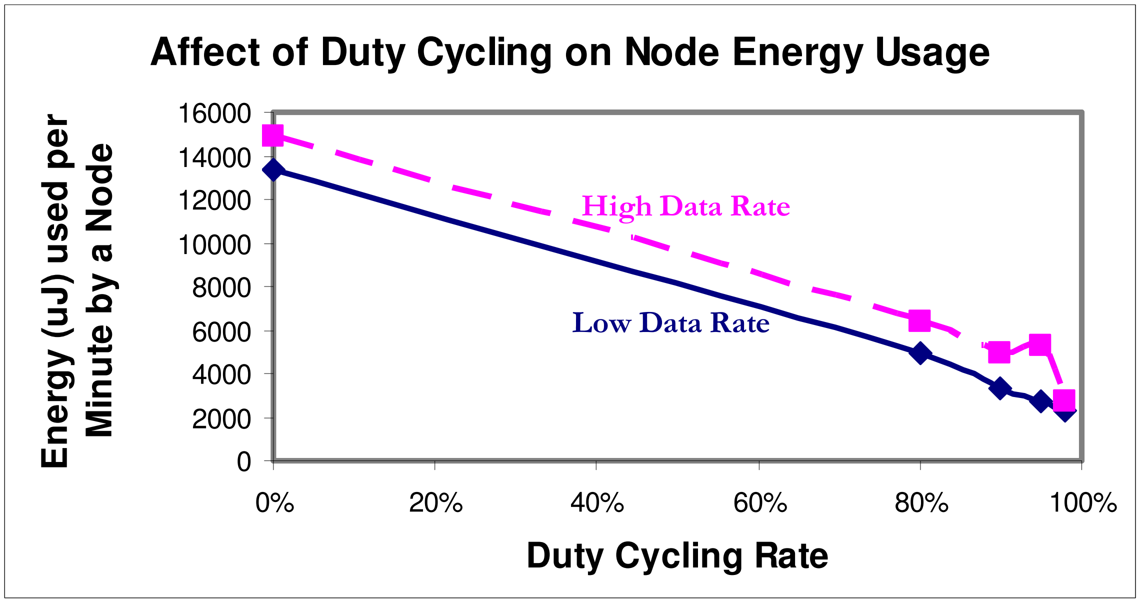

Figure 31: The effects of duty cycling on the amount of energy used per node per minute.

6.2 Other Routers

When designing this fault tolerant middleware one of the goals was for it to work on top of the routing layer. The reason for this is so that if a certain routing layer is considered better for a certain application or environment then that routing layer could also be used with the fault tolerant scheme. If the routing layer chosen provides the default TinyOS routing interfaces then the scheme should work with no modifcation whatsoever. If, for some reason, the routing layer chose to use its own interface then it would be necessary to change the fault tolerant modules a bit so that they can interface properly. In practice, most current routing layers which have been developed for TinyOS do use the proper default interfaces.

In order to test whether it would work with a different router we acquired another popular TinyOS router which is called MINTRoute which is included in the TinyOS distribution [1]. Recall that the original router we used was a distance vector based router which simply tried to find the shortest number of hops to the base station without paying much attention to the quality of the links. MINTRoute is a link state router which calculates a quality rating for different links and will sometimes take a longer path if it has a higher probability of success. In general, we expect that MINTRoute will have a slightly higher success rate than the previous router without fault tolerance but would still benefit a great deal from fault tolerance/

In Figure 32 we show results of experiments done using MINTRoute and compare them to the results using the previous router. We can see that the routers act pretty much the same but there is a bit of a difference stemming from MINTRoute taking link state into account. While both /Route/ and MINTRoute perform well when the DSF is equal to

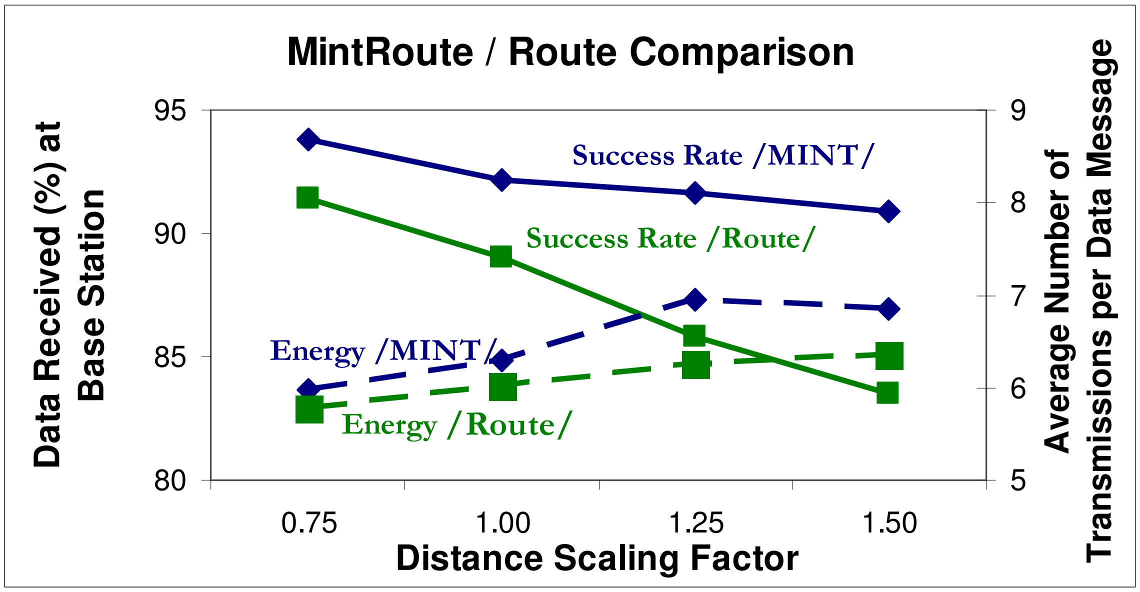

Figure 32: Comparing between MINTRoute and Route. On the left hand y-axis we see the percentage of data that is received at the base station. The two plots which have solid lines align with this axis. On the right hand side is a measure for energy, the two plots with dashed lines align with this axis.

0.75 achieving success rates above 90%, MINTRoute is able to maintain this better as the environment gets harsher and the DSF increases. At a DSF of 1.50 there is close to a 7% different in the performance of the two. This difference can be attributed to the approach that each router takes. While Route attempts to find a shortest path, MINTRoute considers the quality of links and will take a longer path if it means staying on good links. It is clear that there is some benefit to this approach. We see the opposite situation when we examine the energy used. While they are again very similar when the DSF is equal to 0.75 once the DSF reaches 1.50 they become more distinct. In the case of energy it is /Route/ that actually performs better, this makes sense because we know that MINTRoute is sometimes taking longer paths in order to achieve its higher success rate. /Route/ saves on transmissions by taking its shorter paths, even though it costs up to 7% on the success rate.

While it is interesting to see the comparison between the shortest path first approach and the link state approach the really important outcome for our purposes is that we have verified that our fault tolerant layer can be easily transported between different routing mechanisms so long as they properly implement the TinyOS routing interfaces. In this case not a single line of code was changed in the fault tolerant layer in order to achieve the different results between /Route/ and MINTRoute.

6.3 A Larger Network

Throughout this research we have used a constant number of fifty nodes. This number was justified previously because it allowed us to capture all of the important behavior without taking too long for the simulator to run. With fifty nodes we were able to get node depths up to five in many cases and this was usually sufficient for our needs. The problem with increasing the number of nodes is that the time to run the simulation does not increase linearly but goes up exponentially.

Despite the fact that fifty nodes allow us to see most of the interesting behavior, we wanted to run a few experiments with a larger numbers of nodes in order to see what happens as the network depth continues to expand. We observed previously that without fault tolerance the reliability of the network falls off very quickly as the depth increases.

In Figure 33 we see the success rate of nodes in the network at different depths in a one hundred node test. It is apparent that the success rates in Figure 33 drop off much faster than they did in the earlier result shown in Figure 14. There are a number of reasons for this discrepancy. First, while the number of nodes was doubled, the total area only increased by approximately sixty percent. This means that with the nodes arranged more densely there will be more crosstalk and collision affects. Additionally, in Figure

14 we had used four retries while in this case the retry parameter was set to two. We also have small transient failure effects in this experiment that were not present in Figure 14. Finally, we ran this test with a DSF of 1.50 as opposed to 0.75 as in Figure 14.

While the success rates drop faster in Figure 33 than they did in Figure 14 we are still able to see the important trend. Fault tolerance helps even at a depth of one where without fault tolerance 55% of the messages are successful whereas with fault tolerance we receive 92% of the messages. The benefit gets even larger as the depth continues to increase and when we look at, for example, nodes that are of a depth of five we have less than 10% of the messages being heard without fault tolerance but over 50% heard with fault tolerance. This really brings home the point that even with weak parameters (only two retries) and a difficult environment (DSF of 1.50 and high node density) fault tolerance allows us to vastly extend both the number of nodes in the network and the size of the area which they can span.

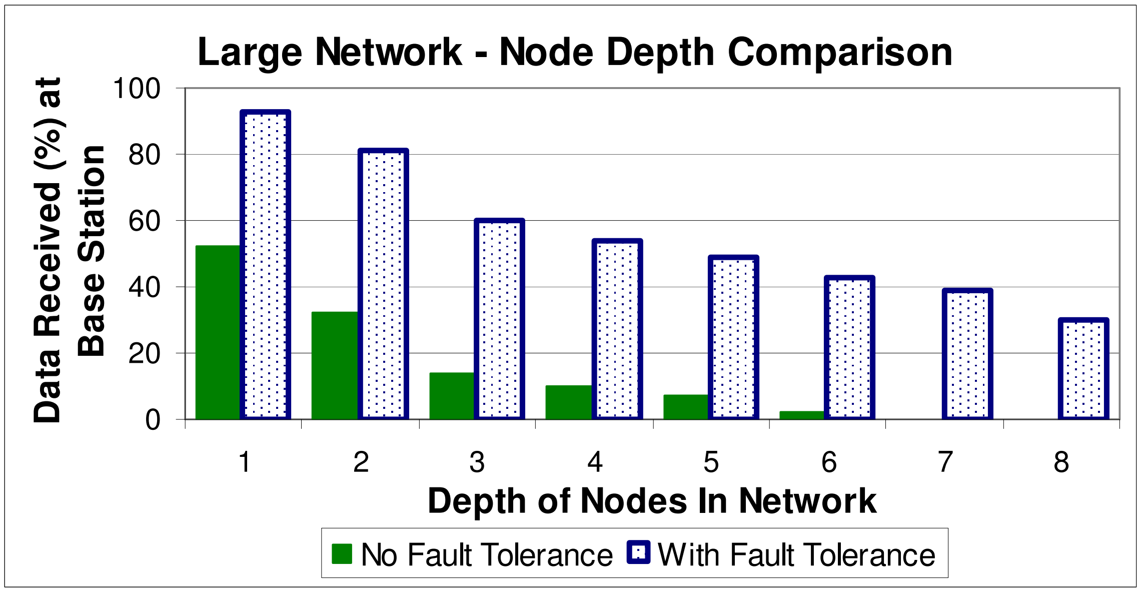

Figure 33: The benefits of fault tolerance in a larger network of one hundred nodes. This data was generated using only two retries and with a DSF of 1.50.

CHAPTER 7

CONCLUSIONS

We have described a series of algorithms that help to provide fault tolerance in sensor network applications that are written in TinyOS. We took the approach of adding fault tolerance in the form of retransmitting and rerouting messages at the routing layer. By implementing at the routing layer we are able to take advantage of information that is available at this layer which may not have been available if we had implemented at a lower layer. This also allows the user to use different TinyOS routing layers and still be able to use the same fault tolerant modules to provide resilience. It also removes the need of each application developer to write their own retransmission and fault tolerance code.

We demonstrated that our technique allowed a vast increase the percentage of data that eventually reaches the base station without adding needlessly to the amount of energy that is being used by each node. Further, we saw that with fault tolerance we are able to expand the size of our networks far beyond what we could use without fault tolerance. We also discussed how in more difficult situations the number of retries that are used can be increased and what affect this has on energy usage.

After these basic findings we looked into making the algorithms learn and remember their past successes and failures. This makes nodes capable of learning which nodes are good routing partners regardless of what the routing layer may have decided when it assigned a depth to each node and without any geographical information. We added transient failures to our experiments and saw which different learning methodologies were better with different rates of transient failures or radio failures.

While in our first experiments events were independent among the nodes, we then looked into what happens when events are correlated amongst nodes that are geographically close by. We saw that this had a detrimental affect on both the success rate of messages and especially the energy used by nodes. This was attributed to a burst of data that happens around the events when they are correlated. We saw that by combining similar messages into one message and scaling the number of retries accordingly, we could better deal with correlated events both in terms of energy usage and success rate of data reaching the base station.

Duty cycling and its affect on success rates and energy were examined by simulating the B-MAC behavior in our experiments. We saw that as the percentage of time that nodes are asleep increases, the energy is reduced and the success rate holds relatively constant until a point is reached where there is not enough time awake to transmit all the data and the success rate rapidly drops off. We also demonstrated the use of the fault tolerance software with another TinyOS routing layer with no changes made to the fault tolerance modules.

CHAPTER 8

FUTURE WORK

While we were able to view many important trends using our simulated experiments the most important future work is to run more comprehensive tests using hardware. Hardware experiments would be very useful in looking at certain behaviors that were difficult to test accurately in the simulator, such as duty cycling. Another area of future work would include a message priority system. This system would scale the number of retries (and consequently, the energy) that are used in trying to retransmit a message based on how important it was to the application. Most mote hardware allows a number of different transmission power modes so that nodes can send weaker transmissions using less energy. There may be interesting fault tolerant and energy saving techniques that could be found by varying the strength of these transmission broadcasts. Finally, experiments that use implicit ACK packets would be useful to validate claims that this would have little impact on the fault tolerance implementation.

REFERENCES

[1] TinyOS Community Forum - http://www.tinyos.net . Documents located in the /tinyos-1.x/doc/ folder of the distribution.

[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. of the 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. of the 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, and 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. 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, 1996.

[8] C. Perkins and E. Royer, "Ad-hoc On-Demand Distance Vector Routing," *Proc. of the 2nd IEEE Workshop on Mobile Computer Sytems and Applications*, pp. 90-100, February 1999.

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

[10] R.C. Shah, and J.M. Rabaey, "Energy aware routing for low energy ad hoc sensor networks," *WCNC2002. 2002 IEEE* , vol.1, .pp. 350- 355, Mar 2002.

[11] R. Jurdak, P. Baldi, and C. V. Lopes, "Energy-Aware Adaptive Low Power Listening for Sensor Networks", in *proc. 2nd International Workshop for Networked Sensing Systems*, June 2005.

[12] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," *Proc. of the 2nd international conference on Embedded networked sensor systems*, pp. 95-107, 2004.

[13] CE Perkins, and P Bhagwat, "Highly dynamic destination sequenced distance vector routing for mobile computers" in *Proceedings of ACM SIGCOMM,* pp. 234 - 244 1994.

[14] T Clausen, P Jacquet, A Laouiti, et al. "Optimized Link State Routing Protocol", *IEEE INMIC*, pp. 95-107, 2001.

[15] V. Park, and M. Scott Corson, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks," *INFOCOM '07,* pp. 1405-1413, 1997.

[16] J.J. Garcia-Luna-Aceves, and M Spohn, "Source-Tree Routing in Wireless Networks", *Seventh International Conference on Network Protocols ICNP'99,* pp.273-282, 1999.

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

[18] U. Malesci and S. Madden, "A Measurement-based Analysis of the Interaction between Network Layers in TinyOS", *Proc. of EWSN*, February 2006.

[19] C. Intanagonwiwat , D. Estrin , R. Govindan and J. Heidemann, "Impact of Network Density on Data Aggregation in Wireless Sensor Networks", *Proc. of ICDCS'02*, pp.457-458, July 2002.

[20] C. Karlof, N. Sastry, D. Wagner, "TinySec: a link layer security architecture for wireless sensor networks", *Proc. of the 2nd international conference on Embedded networked sensor systems*, pp. 162-175, November 2004.

[21] Intel Research – Sensor Nets / RFID
http://www.intel.com/research/exploratory/wireless_sensors.htm