SIMULATING UNDERWATER SENSOR NETWORKS AND ROUTING ALGORITHMS IN MATLAB

by

Michael J. O'Rourke

A Thesis Submitted to the

Office of Research and Graduate Studies

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

School of Engineering and Computer Science
Engineering Science

University of the Pacific
Stockton, California

2013

SIMULATING UNDERWATER SENSOR NETWORKS AND ROUTING ALGORITHMS IN MATLAB

by

Michael J. O'Rourke

APPROVED BY:

Thesis Advisor: Elizabeth Basha, Ph.D.

Committee Member: Carrick Detweiler, Ph.D.

Committee Member: Ken Hughes, Ph.D.

Department Chairperson: Jennifer Ross, Ph.D.

Interim Dean of Graduate Studies: Bhaskara R. Jasti, Ph.D.

DEDICATION

This work is dedicated my family, friends, thesis committee, and the staff of the University of the Pacific School of Engineering and Computer Science. The endless support I've received from all parties allowed me to complete this and without their help I'd have given up long ago.

ACKNOWLEDGMENTS

I'd like to acknowledge my committee: Dr. Basha, Dr. Detweiler, and Dr. Hughes. Dr. Basha, chair of the committee and my advisor, regularly offered keen insight and gave me freedom to approach this my own way. Her patience has been admirable, and I appreciate that I had the opportunity to work with her. Dr. Detweiler challenged me to carry this project forward and helped me identify ways to be more productive in development. Dr. Hughes and Dr. Detweiler helped me understand where weak points have been, and with the members of my committee I believe I've done something to be proud of.

I would also like to thank the National Science Foundation for supporting the research (CSR #1217400 and CSR #1217428) as well as the School of Engineering and Computer Science.

I'd also like to thank my friend Adam Yee. Whether it was talking through my ideas late at night, explaining the system in different ways, or simply justifying my decisions, Adam was always available. With his assistance, I was able to keep my goals realistic and intentions clear.

Simulating Underwater Sensor Networks and Routing Algorithms in MATLAB

Abstract

by Michael J. O'Rourke

University of the Pacific
2013

Underwater networks are a field that has been gathering attention. Land-based methods of network construction from discovery to advanced routing are all well established. Due to the unique constraints of operating in an underwater environment, many of these tried-and-true approaches need modification if they function at all. Peer discovery and low-level networking have been dealt with by previous research. In this thesis we describe and evaluate eight possible routing schemes with different system-knowledge requirements. We show that with a minimal set of information it is possible to achieve near-optimal results with energy costs considerably lower than centralized optimal algorithms.

We demonstrate this by constructing and evaluating a custom simulation environment in MATLAB. This is implemented in a mixed procedural and array-centric approach. Simulated networks are structured on a line topology. All nodes a spaced along the horizontal axis at a random depth. It is assumed that neighbor-discovery has been completed before the simulation starts, and all nodes have access to a global list of connected neighbors. We

demonstrate the effectiveness of distributed algorithms in an ideal environment, leading to the conclusion that near-optimal results can be achieved with local information only.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

List of Algorithms

Chapter 1: Introduction

Just over 70% of the planet is covered in water. Very little of that is explored, and we likely understand even less. One way researchers are approaching this is by innovating with underwater robots and networks. In this thesis we define a simulation environment that describes an underwater network where nodes are able to efficiently surface to use radio communication. Using this environment we determine paths for radio communication on a line topology in a multimodal system. We implement a total of eight different algorithms to facilitate decisions about communication.

The agents emulated in this thesis have depth control capabilities and two wireless modems. The first modem is an acoustic device that allows any two nodes within range to communicate underwater, but at a high energy cost. The second is a radio that can not be used underwater, but requires relatively little power. The acoustic and radio devices modeled in this thesis are described in Table 1, with values taken from [1]. The depth adjustment system described in [1,2] consumes a large amount of power compared to either modem.

	Acoustic [2–4]	Aerocomm AC4790
Transmit Power	5W, 113.6mJ/bit	1W, 0.16mJ/bit
Data Rate	22b/s	7.2kb/s

Table 1: Summary of acoustic and radio modem attributes

Our primary concern is then: how do we pass a message across the network in an efficient way? As shown in [1, 5], there is a clear cutoff of when the cost of rising will be less than the cost of forwarding a large message acoustically. In reference to that, the algorithms defined in Chapter 4 provide a method of selecting who will rise and participate in radio-message forwarding. We determine that distributed algorithms using only local information can perform at or near the level globally optimal algorithms for determining members of the radio path. This thesis contributes to the field of underwater networks by demonstrating the effectiveness of greedy routing schemes that use locally available information, and by providing a simulator in which they can be tested.

1.1 The Problem and Approach

The devices we model in this thesis is based on AquaNodes. Shown in Figure 1 are

Figure 1: Picture of an AquaNode [5]

two AquaNodes, one complete in its water-proof casing and one exposed. In addition to the modems mentioned earlier, these devices have multiple processors, a range of sensors, and a depth control system. We simplify this by assuming the more power-consuming processor is always on; the sensors are not implemented within this thesis.

The radio on each system is not usable underwater. As presented in Table 1, radio communication is both faster and less expensive. In our context, the problem with radio communication is the *cost of surfacing*; using the depth adjustment system. The average cost of motion, ascending or descending, is determined to be 15000mJ/m or 15 Joules/meter. All other energy costs are in the mJ, or in some cases μJ range. Rising then dominates cost until message size increases above a threshold.

We assume in this thesis that a large collection of data, well exceeding the point at which sending it all acoustically is more expensive than the average cost of rising, is ready to be forwarded. It is thus our task to determine which nodes in the network will surface to participate in forwarding radio messages.

Briefly we mentioned having a set of algorithms for determining radio path. An example of this decision making process can be seen in Figure 2. The algorithms we implemented can be separated in to two main groups: acoustic centric and radio centric. Acoustic centric algorithms rely solely on single-hop acoustic network links to determine the radio path. Radio centric algorithms will allow for multiple acoustic hops to determine a radio path, and extend their path-formation decisions to include all neighbors within radio range. Our assumptions are:

- Perfect communication in the system,
- nodes are constantly active,

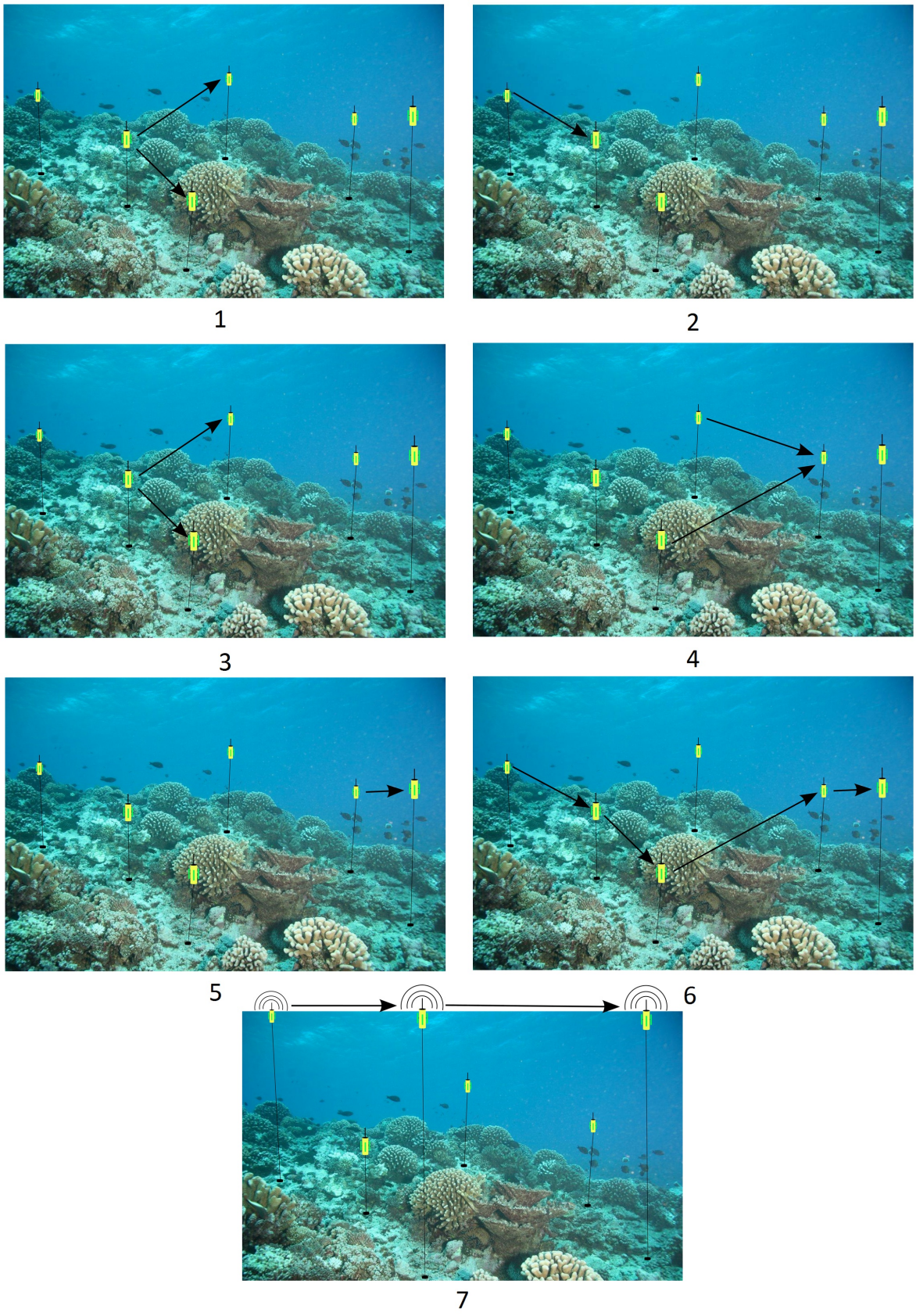

Figure 2: AquaNodes finding a radio path

- limitless power for each node,
- peer discovery is completed before the simulation starts,
- and that nodes will only move vertically.

We then seek to determine the best method of choosing a radio path. All of the algorithms are described in detail in Chapter 4, and the experiments we conduct to determine the best methods are discussed at length in Chapter 5.

1.2 Overview of Experiments

There are a total of four basic experiments for testing network models, and two for characterizing the environment. For characterizing the environment we run a single-algorithm at default system settings on two different systems. The first system is running Windows XP, and demonstrates performance on previous generation systems. The second system is running Ubuntu Linux and has much higher capabilities (memory, processing, etc.). The systems and experiments are presented in further detail in 3.4.

The four algorithm/model tests are: all default settings, default settings but changing node-placement rules, default settings and changing acoustic range, and default settings while changing radio range. The first experiment details typical behavior of the system across all algorithms. The second experiment explains the effect of node placement on algorithm performance.

In acoustic-range testing, the range is set to the minimum placement between nodes and then expanded several times beyond maximum distance between neighbors. A similar track is followed for radio-range testing. Radio testing constrains itself to default acoustic range, expands out beyond to typical range, and is finally set to several times typical range.

All of the range testing relies on typical node placement which has a random aspect to it, but guarantees that all nodes have at least two acoustic neighbors (and at most six).

The basic all-algorithm test and node placement tests are run on all algorithms. The range tests are run on a subset including *Greedy Furthest Radio* (4.2.1), *Greedy Shallowest Radio* (4.2.2), *Greedy Look-Ahead* (4.2.3), and *Min-Hop Shallowest* (4.2.6).

1.3 Thesis Outline

This thesis is composed of five additional chapters. In chapter 2 we discuss preliminary work in underwater networks. With chapter 3 we describe and characterize the custom simulation environment we created to support this work. The inner workings of the environment are discussed there, and data are presented that describes environment performance on different platforms. Going to chapter 4 all eight algorithms are described, characterized over one-thousand unique topologies on sizes ranging from 25 to 100 nodes. Basic statistics are presented as well as sample graphs demonstrating the behavior of each algorithm on a ten-node network.

Chapter 5 describes all experiments at length and presents the results. Concluding remarks are offered in 6, and followed by an appendix containing the code for all algorithms and the environment.

Chapter 2: Related Work

Our system takes advantage of multiple communication methods to within an underwater network. We develop an idealized simulation environment that eliminates many of the challenges of networking, and assume peer discovery has already been completed. As we function on an abstract level rather than handling specific issues of implementing networked systems, we draw from algorithmic studies and rely on values from work done with physical systems.

2.1 Wireless Underwater Networks

It is common for sensor networks to rely on gateway nodes to handle large amounts of data over long ranges [6–8]. An example of this is the SeaWeb [9] system. SeaWeb used nodes at the surface to support communication with the outside world, and for localization within its own network via GPS. Research has been done that demonstrates the practical nature of using gateways in underwater networks [10]. Zhou et al. used linear programming to optimize the placement of these gateways to minimize power and delay [11].

What could be considered a flaw of these surface-gateway systems is that the acoustic modem limits communication. All nodes must transmit acoustically to have data forwarded out of the underwater part of the network. This places a natural limit on the rate at which data can be extricated; in an ideal system data could be retrieved at the maximum rate of the modem, but a real system would face packet loss. To mitigate this problem, work has been

done to introduce underwater vehicles to these systems and retrieve data as policy dictates. These vehicles collect data along a path and occasionally rise to send data via radio [1, 12].

The system we implement is different in that each node is capable of surfacing to send its own data or relay transmissions from its neighbors. Issues associated with acoustic gateways are removed in this scenario. The drawback of this approach is that nodes could be moved from their correct location for sensing, and the power-cost of rising needs to be factored in to routing decisions. For this paper we explore different approaches for determining radio paths on a line topology, attempting to minimizing energy consumption.

To that end we implement optimal path finding algorithms, examine greedy approaches, and compare the results of these. The greedy approaches are similar to greedy geographic routing in land-based systems which are near-optimal in dense networks, and varying implementations have been demonstrated to improve performance in challenging and dynamic environments [13, 14]. Given the high performance of greedy methods in land-environments, we are motivated to explore their loose analogues in the underwater arena.

2.2 Multimodal Communication

Land-based sensor networks have been exploring multimodal communication for years. For example, in [15, 16] the authors describe gateway devices that combine short range and long range communication. Just like underwater systems, the in-network modem and gateways become choke-points. Chen and Ma describe MEMOSEN [6], and demonstrate the effectiveness of mobile multimodal systems in sensor networks. MEMOSEN uses clustering to structure networks. Within a cluster, a low-power radio such as bluetooth is used to facilitate data transfer. Between clusters and out of the system use a longer-range/higher-power device, with designated gateways.

Our system does not deal with gateways, and intentionally allows for *any node* to rise and send its own data. In that regard, we see the idea of opportunistic mode changes in [17]. Chumkamon et al. explore vehicular networks that arbitrate between modems based on convenience and availability. We see the greatest connection with our own research by combining the works of [18] and [6]. Lin et al. use a decentralized system of low-power communication coordinators to determine which network agents will use higher-power devices to send or relay data. MEMOSEN describes a distributed mobile system without strict rules on who will engage in high-throughput communication.

We differ from and extend these by: moving to underwater systems, changing communication parameters (i.e., our high-throughput is our low-power device), and we have a set decision policy for determining who will participate in "elevated" communication.

Chapter 3: The Simulation Environment

In this chapter we discuss the software we developed to simulate aquatic networks. We start be explaining why we chose to develop in MATLAB. Then we discuss the two versions of the simulator we implemented, an object-oriented approach and a procedural approach respectively. Originally we used the object oriented approach, but abandoned it in favor of the procedural approach due to the difficulties introduced into the debug process that were not relevant to this thesis. Then we detail the basic structure of the system, define performance metrics of the environment and models, and close with a discussion of simulation performance on two platforms.

3.1 MATLAB

The environment is implemented as a set of MATLAB scripts and functions. There are a several benefits to this decision including: ease of development, industry acceptance, native support for large numbers, and efficient storage and handling of large data sets. Using a high level environment like MATLAB makes development easier through simple syntax and advanced libraries. For example: functions for determining distance between all points in a set already exist, and so do an entire suite of statistical analysis tools. We did not have to implement any of these fundamental background tasks. With the tools provided and high level interface, MATLAB was the natural choice as it allowed for rapid prototyping and analysis. Our only concern of development was the *functionality* of the system itself.

The rest of this section describes the two implementations of the simulation environment, their weaknesses, and strengths.

3.1.1 Object Oriented Environment. The first implementation of the environment used MATLAB's object-oriented (OO) paradigm. This was a natural choice as it allowed for storing private meta-data and associating actions with the agents performing them. The environment was described in four basic classes. Those classes were:

- Sensor

  A skeleton class that contained an ID string, a value read from the environment, and basic functions to poll for new data. The intention was that a more specific subclass would be made that could properly emulate the behavior of a given sensor, including the power requirements and format in which it provides data.

- Communicator

  A basic implementation of a communication modem. This included send and receive queues, power requirements for sending and receiving, and success rates that served as a proxy for the type of device and medium through which it transmitted.

- Node

  The Node class was a container of Communicators and Sensors, and represented the basic "agent" in the simulation. Nodes stored information on position, a list of neighbors and their positions, their ID value, and their three-dimensional coordinates. The algorithm being run to make routing decisions was stored within each node, and all necessary support functions for the decision making progress. Nodes provided information on messages sent and power consumed to external classes and scripts.

- Network

  The container of containers. The Network class contained nodes, maintained a regularly changing connectivity matrix and list of positions, and stored simulation meta-data. As one large batch process all messages sent were processed in a message handling function that determined whether or not a message would arrive at any given destination. Total messages sent and received were stored as well as power consumed, distance travelled, and time to execute the algorithm per node.

The OO-approach had benefits in that data could be stored in an intuitive hierarchy and it was easy to modify and expand, with or without thorough planning.

When a network was instantiated, it would store the positions of nodes and their communication paths based on location and range information. As nodes were created they were provided an ID value, their position, and the list of their neighbors. Communication modems and sensors were populated. Nodes contained a reference to the MATLAB version of the code they would run in a physical implementation. It was observed that in some topologies where four or more nodes were placed very closely together, such that they each had several acoustic neighbors with a lot of overlap, an issue of message duplication would occur. Members of these dense clusters would defy forwarding rules and would attempt to send hundreds or thousands of duplicate messages. To combat this problem we implemented a "sent messages list." Any time a message was sent it would be added to a nodes "sent list." If a message was in the list, it would be dropped without being passing through any sending logic. This did not solve the problem however. The only reliable way we could remove the issue was by moving away from the use of objects. Objects are more realistic, and in a "real" setting this could be a potential issue.

This work is abstract, not a low-level system that needs to precisely follow real-world issues. It was not possible to reliably obtain data and ensure a fair comparison between algorithms using this approach. Out of necessity the OO environment was abandoned.

3.1.2 Procedural Environment. The final version of the environment is procedural with data in correlation arrays. The strengths of the implementation are its simplicity, ease of adding features, improved maintainability, and a gain in performance acquired by reducing the number of references. A collection of scripts establish and run the system with helper functions where non-damaging change simplifies calculations. We separate scripts and functions into three categories as follows:

- Initializers

  These are single purpose scripts. They initialize a data field (node positions, message queues) and define a constants, such as radio range, that will be used throughout the simulation.

- Modifiers

  Generally implemented as functions, they make temporary partial-copies of matrices and return the result. A prime example of this is the handling of messages queues. There is a function called `add2buffer` that takes as parameters the message queue, its read and write pointers, and the new message. Returned is the modified queue and pointers. Modifiers are helper functions, and sometimes scripts, that manipulate storage structures rather than just using them.

- Run-Time

Running the simulation and the algorithmic-helpers. The process of generating messages, determining how to respond, and constructing the path radio messages will follow, are all run-time tasks. Along with the aptly named `Sim_Run` script, are the scripts for the routing algorithms.

We show the flow of events in Figure 3. Constants are defined which include the number

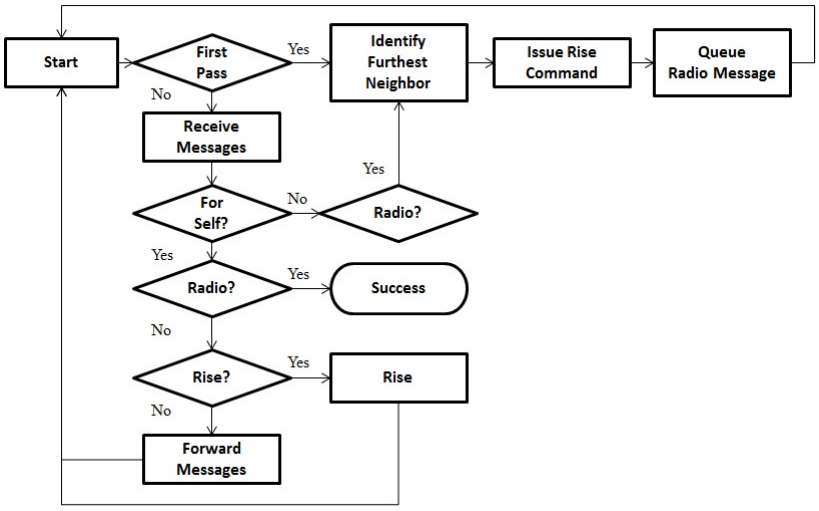

Figure 3: Chart of Simulation Structure

of nodes, queue length, communication power and range, and so on. Node positions are generated on a line topology following placement rules set when constants were defined. Network connections are determined based on positions and range values. Message queues only require the number of nodes and queue length, which is why it is shown to be a separate process path.

Once the environment is prepared, we enter into a loop that is ended on two conditions: successful message forwarding, or simulation timeout (indicative of an error). The loop nodes work through simply responds to messages. The first node starts the process, the

rest of the system behaves as a reaction to that initial action. Messages are either dropped, forwarded, or reacted to locally (rise or end-condition). This basic process is shown in Algorithm 1, to provide an alternative view.

Algorithm 1 Basic Sequence of Events

```
constantInit
topologyInit
connectivityCreate
CommInit
create new radio message
send first rise packet
repeat
    receive messages
    determineNextAction     ▷ Response is determined by message contents and model.
    perform action          ▷ Drop the packet, forward the packet, or rise.
    send messages
until radio message arrives at destination
```

3.2 Simulation Structure

It is important to understand both the design and flow of the system. To that end we will describe the basic nature of the system and then move to what the simulator does and in what order.

The simulation environment is based on discrete events. Specifically, it operates on acoustic-communication windows. Each window lasts approximately 4N seconds of simulation time, where N is the number of network nodes. Each agent has the opportunity to read messages sent from the previous iteration, send messages, and move on the vertical axis. Any action that can be performed is assumed to be done during an event, though the environment could be modified to support a pre-/post-event phase for longer actions to occur. This event-cycle simplified implementation; expanding the simulator with out-of-event

processing would restore some realism to the system. Armed with this understanding of what the simulator does, and the description is basic components, we move on to simulation flow.

As shown in Algorithm 1, the simulator initializes the environment and launches a run script. The result of the simulation is a collection of matrices that contain data on position, messages sent and received, energy consumed, and time taken. The process of running the simulation requires information on position, message queues, and communication modems (range, power). Initialization is separated across four files: `constantInit`, `topologyInit`, `connectivityCreate`, and `CommInit`.

The script `constantInit` checks to see if required constants have been defined already, and sets them if they do not exist. These values include but are not limited to the range and power of the radio and acoustic modems, the maximum length of a message queue, the energy cost of different actions, and the number of nodes to generate. Topologies are either generated or imported using the `topologyInit` script. This script checks to see if positions have been defined, and if not it generates a random line topology within defined constraints. From the topology, `connectivityCreate` uses modem ranges to create a connectivity matrix for each communication modem. The connectivity matrix is an NxN logical matrix, where true represents a connection. `CommInit` creates message queues, implemented as ring buffers, using the number of nodes and the queue length.

After initialization `Sim_Run` uses constants defined in `constantInit` and the structures from the other initializers to determine the radio path, simulate the message-passing process, and generally emulate idealized behavior of individual nodes.

3.3 Environment Performance Metrics

In this section we define how we measure environment performance. Aside from whether the result is correct or not, we can determine performance by how much time is required for networks of different size, and how much memory the system uses. Naturally, when comparing implementations the better choice is the one that minimizes necessary resources. If two have identical spatial requirements yet one is clearly faster, it is the superior implementation, *mutatis mutandi* for fixed time and differing space. Following is a description of how we measure time and memory requirements in MATLAB on Windows and Linux platforms.

3.3.1 How Time is Measured. Built in to MATLAB are the functions `tic` and `toc`. These respectively start a timer and calculate the approximate number of seconds that have elapsed since the timer was started. This is the simplest way in which to monitor the performance on a given system. It is platform independent, which makes it a natural choice.

3.3.2 How Memory is Measured. MATLAB has two methods of reporting memory usage. One is a Windows-only command called `memory`, and the other is `whos`. The former reports out byte-usage of the entire platform, and will provide information relating to memory pages and other low level information. The latter iterates through all variables in the current workspace and reports the number of bytes each item consumes. We use `whos` for this project. Using `memory` or using an operating system tool such as `top` on Linux provides information about the entire system. For a true comparison, we use `whos`, as this will avoid any background tasks that MATLAB undertakes during simulation.

3.4 Performance Results

In this section we describe a set of experiments conducted to show the performance of the simulator and model. The experiments included are:

1. Default Settings, 10 Nodes, *Greedy Shallowest Radio* algorithm

   This algorithm has each node search for its shallowest neighbor closer to the destination than itself. That neighbor will be who it forwards radio messages to. This is described more fully in 4.2.2.
2. Default Settings, 50 Nodes
3. Default Settings, 100 Nodes

Default settings guarantees at least two acoustic neighbors for each node, and no more than 6 acoustic neighbors. The two systems are described below in Table 2.

	OS	Memory	CPU	MATLAB
System 1	MS Windows XP, SP3	2GB, 800MHz	Intel E6550, 2.33GHz	RS2010a
System 2	Ubuntu 12.04 (3.2.0-32)	8GB, 1333MHz	Intel E31240, 3.3GHz	RS2010a (64-bit)

Table 2: System information for the two test platforms

3.4.1 Environment Performance. The results from the performance tests for the legacy Windows platform and the Linux platform can be found in Tables 3 and 4, respectively. The memory usage posted is without the base-level of the MATLAB application. The values are *only* what simulation variables required, before and after running. At first glance, we observe the rate at which memory and time requirements increase. The result for 100 nodes on Linux, post-run, is just over 925KB. Moving in to larger numbers of nodes will necessarily take longer, and the space requirements will grow. From 10 to 100

	Number of Nodes		
	10	50	100
Memory Pre-Run (B)	3704	37904	125672
Memory Post-Run (B)	20787	211672	700828
Run Time (s)	0.24	3.45	12.11

Table 3: Windows Platform

	Number of Nodes		
	10	50	100
Memory Pre-Run (B)	5880	64088	218432
Memory Post-Run (B)	39419	311475	948211
Run Time (s)	0.16	2.39	8.36

Table 4: Linux Platform

nodes we observe a leap from 38KB to 925KB. For a linear progression, we would expect 50 nodes to be nearly a half-way point between the others. As this is not the case, we can reasonably conclude the memory consumption is exponential with network size. This can be understood simply as "every matrix that stores node-data must add a new row/column" and that not all collections will grow the same way.

It is interesting to note that the Windows environment consistently reports using approximately 60% of the space used in Linux. We can observe that between platforms, MATLAB's `cell` type has different memory requirements. Without information on the difference between the Win32 and glnx64 implementations of MATLAB, we can not conclude precisely why. Despite the difference in memory the time required to complete a full simulation is faster in Linux, as shown in Tables 3 and 4. Due to the increase in speed we proceed forward with the Linux platform for the duration of this thesis.

Chapter 4: Algorithms Implemented

In this chapter we discuss the algorithms we implemented. There are a total of eight different algorithms separated into two categories. The categories are "acoustic-centric" and "radio-centric" algorithms. An algorithm is considered acoustic-centric if routing decisions are made from information on acoustic neighbors. So, naturally, a radio-centric method is one that makes decisions based off of radio neighbors.

All of these algorithms rely on the connectivity matrix defined in 3.2. In short, a logical matrix where *true* represents a connection, and the connections are determined by the distance between nodes and the range of the active modem. Further, all of the algorithms assume radio range is greater than or equal to acoustic range. This is not a requirement of the environment, it is only a detail of how they were implemented. This was done to simplify implementation, and with relatively minor changes each algorithm could be modified to be range-agnostic. The final requirement imposed, that is necessary for a successful simulation, is that each node have at least one acoustic neighbor. This automatically guarantees a minimum of one radio neighbor because if two nodes are within acoustic range, they are guaranteed to be within radio range. For any algorithm, the general sequence of events is: receive a radio message, identify the next-hop in the path, issue a rise command, and forward the radio message after a short delay. An example of this can be seen in Algorithm 1, which shows the generic process from start to finish.

The rest of this chapter is separated as discussions of acoustic algorithms and radio

algorithms respectively. The algorithms are discussed and sample data for each is provided.

4.1 Acoustic-Centric Algorithms

There are two acoustic-centric algorithms. A furthest-neighbor approach and a shallowest-neighbor approach. These are called *Greedy Furthest Acoustic* and *Greedy Shallowest Acoustic*, respectively.

4.1.1 Greedy Furthest Acoustic. When a node receives a radio message it will use the connectivity matrix to determine its furthest connected neighbor. A rise command is sent to that neighbor, and after a short delay the radio message is re-broadcast. Rise commands are acoustic messages. They contain no data; the fields of the packet are destination, source, and a protocol/command field corresponding to "rise," as can be seen in appendix A.3.3 and in locations throughout A.2. This process repeats until the radio message reaches its destination, like shown in Algorithm 2.

Algorithm 2 Overview of Greedy Furthest Acoustic

```
loop
    receive radio packet P
    if P.Destination ≡ Self then
        exit
    else
        queue P for delayed-transmission
        find furthest acoustic neighbor          ▷ See Algorithm 3
        send rise command
    end if
end loop
```

It is important to understand that except for 4.2.5 and 4.2.6, all algorithms behave in this same way. The only change between algorithms is how next-neighbor's are selected.

Algorithm 3 Determining the furthest connected neighbor; excerpt from MATLAB code

```
function id = greedy_farthest_algorithm(self_ID,conn,dst_id,Node_Pos)
% pass in the destination ID, and the row of the present node
% in the connectivity matrix, and node positions
    id = 0;
    Conn_quick_ref = [];
    for i =1:length(conn)
        if i == self_ID;
            continue
        end
        if conn(i)
            Conn_quick_ref = [Conn_quick_ref i];
        end
    end
    Pos = zeros(length(Conn_quick_ref)+1,1);
    for i = 1:(length(Conn_quick_ref))
        Pos(i+1,:) = Node_Pos(Conn_quick_ref(i),1);
    end
    Pos(1,:) = Node_Pos(dst_id,1);
    dist = squareform(pdist(Pos));
    tmp_distance = inf;
    for i = 1:length(dist(:,1))
        if i == 1
            continue
        end
        if dist(i,1) < tmp_distance
            % if the distance is the least we've found
            % select that index and distance
            % that index can then be used to select from the quick ref.
            tmp_distance = dist(i,1);
            id = i;
        end
    end
    if id == 0
        error('Could not find any neighbors, which is not likely.\n')
    else
        id = Conn_quick_ref(id - 1);
    end
end
```

Figure 4 shows a sample topology and the path Greedy Furthest Acoustic created. If each node has only one forward-neighbor available, one neighbor closer to the destination, then this algorithm would require all nodes to rise. Nodes being spaced evenly near the edge of acoustic range would create a worst-case topology for this algorithm, as this is the definition of having only one forward-neighbor. Nodes placed close together or acoustic range long enough to allow for multiple forward-neighbors allow this algorithm to see significant improvement in performance. Averaged over 500 runs, the performance of the model implementing this algorithm is summarized in Table 5.

The fields of Table 5 are: Average Distance, Average Depth, Average Energy, and Average Time. The distance field represents the amount of total distance travelled by nodes in the network averaged across all nodes including those that do not participate in routing. Depth is the average starting depth for all nodes. Energy consumption is averaged for all nodes in a network, with most energy being consumed by node movement. Average time

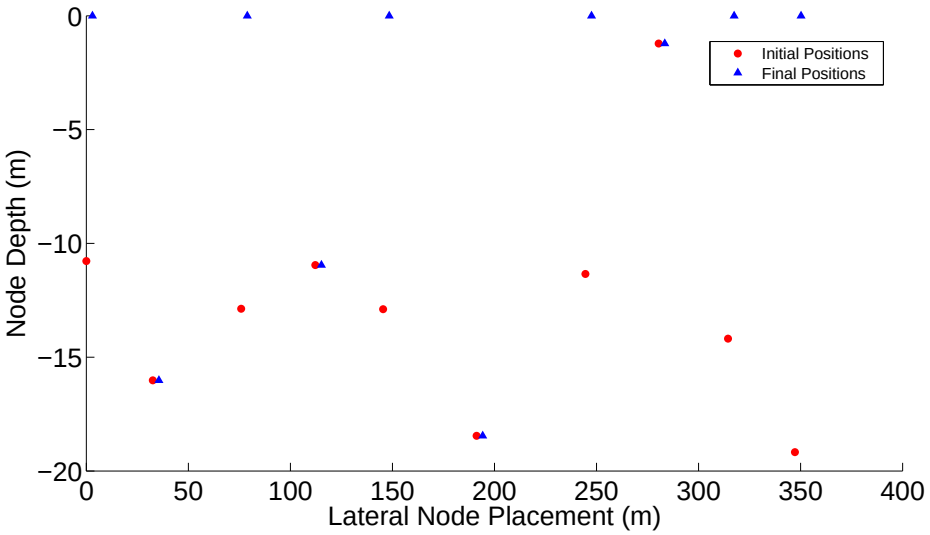

Figure 4: Graph of Start and End positions for Greedy Furthest Acoustic

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	6.5131	6.7744	6.7548	6.6132
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy (J)	90.53	89.50	88.57	88.26
Avg. Time (μs)	62.2138	62.6475	61.9471	61.8014

Table 5: Sample Statistics for Greedy Furthest Acoustic

represents the amount of time it took for each node to complete a single iteration through the simulation.

Immediately obvious is that the average depth is always around -10 meters. This is expected, as the range is from -20m to 0m depth. The depth a node begins at is selected randomly; a random number from 0 to 1 is generated, and multiplied by -20. After enough topologies are generated, we would observe a relatively even distribution of nodes from 0m depth to -20m depth, for an average of -10m. Somewhat surprising to note is the relatively small change in average distance and energy. This implies that the load on all participating nodes, and all nodes in the network, is agnostic to network size. It is clear to see that this algorithm's performance scales well. We observe that 4.1.2 and all of 4.2 have much lower

energy requirements than *Greedy Furthest Acoustic*.

4.1.2 Greedy Shallowest Acoustic. This approach requires nodes to determine their shallowest neighbor that is closer to the destination than themselves, as described in Algorithm 4. The structure and flow is identical to Algorithm 2, but selecting different

Algorithm 4 Overview of Greedy Furthest Acoustic

```
loop
    receive radio packet P
    if P.Destination ≡ Self then
        exit
    else
        queue P for delayed-transmission
        find shallowest acoustic neighbor          ▷ See Algorithm 5
        send rise command
    end if
end loop
```

attributes. Specifically, instead of finding our furthest neighbor we identify all neighbors closer to the destination than ourselves and select the shallowest of that set to rise. The best case of this algorithm is when the furthest acoustic neighbor is also shallowest. The worst cases for this approach are monotonically increasing depth and all nodes having only one forward neighbor. Both of these result in an all-rise scenario. Figure 5 shows sample position data for the Greedy Shallowest Acoustic algorithm. Performance data is summarized in Table 6.

There is an immediate performance increase when selecting *Greedy Shallowest Acoustic* over *Greedy Furthest Acoustic*. As a general rule, the energy cost of rising is greater than most acoustic messages. This algorithm will minimize the amount of travel required on a per-hop basis (average case), but is limited to selecting a neighbor within acoustic

Algorithm 5 Determining the shallowest connected neighbor; excerpt from MATLAB code

```
function id = greedy_shallowest_algorithm(self_ID,conn,dst_id,Node_Pos)
    id = 0;
    Conn_quick_ref = [];
    for i =1:length(conn)
        if i == self_ID;
            continue
        end
        if conn(i)
            if i == dst_id
                id = i;
                return
            else
                Conn_quick_ref = [Conn_quick_ref i];
            end
        end
    end
    Pos = zeros(length(Conn_quick_ref)+2,3);
    for i = 1:(length(Conn_quick_ref))
        Pos(i+2,:) = Node_Pos(Conn_quick_ref(i),:);
    end
    Pos(1,:) = Node_Pos(dst_id,:);
    Pos(2,:) = Node_Pos(self_ID,:);
    dist = squareform(pdist(Pos(:,1:2)));
    depth_iter = -inf;
    for i=3:length(dist(1,:))
        if (dist(i,1) < dist(2,1)) && (depth_iter <= Pos(i,3))
            depth_iter = Pos(i,3);
            id = i;
        end
    end

    if id == 0
        error('Could not find any neighbors, which is not likely.\n')
    else
        id = Conn_quick_ref(id - 2);
    end
end
```

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	6.0429	5.8105	5.7679	5.8028
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy (J)	90.25	87.73	87.71	87.65
Avg. Time (μs)	63.2898	63.2063	62.2386	61.7403

Table 6: Sample Statistics for Greedy Shallowest Acoustic

range. For all network sizes, *Greedy Shallowest Acoustic* requires less motion and power. This performance relationship will be seen again with algorithms 4.2.1 and 4.2.2. Required execution time is comparable to *Greedy Furthest Acoustic*; the longer times are a result of accepting more hops in a network to minimize the amount of distance travelled by an individual.

4.2 Radio-Centric Algorithms

All other algorithms implemented are based on a node radio capabilities. The six implemented algorithms are: *Greedy Furthest Radio, Greedy Shallowest Radio, Greedy*

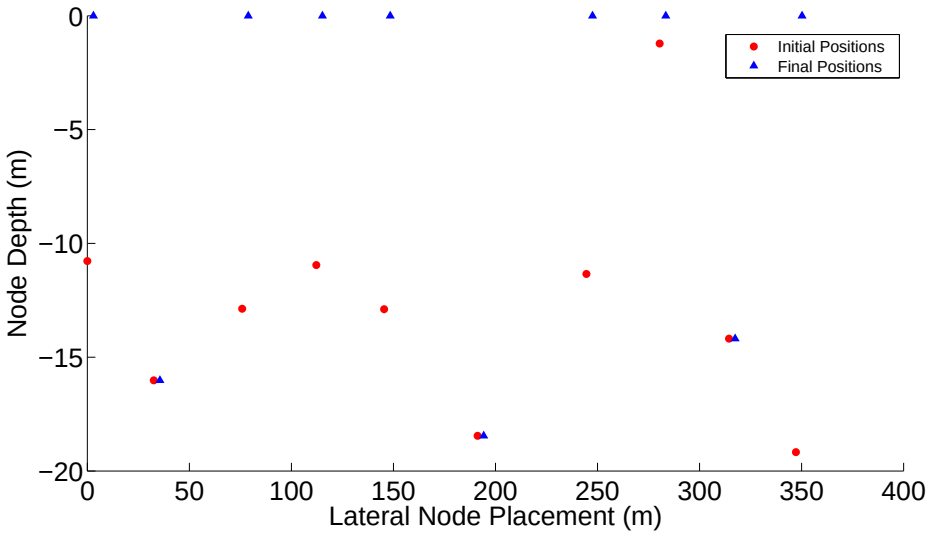

Figure 5: Graph of Start and End positions for Greedy Shallowest Acoustic

Look-Ahead, Greedy Look-Back, Min-Hop Furthest, and *Min-Hop Shallowest*. The first two rely only on information about immediate neighbors, the second set of two use information about their neighbors and their neighbors-neighbors. The final two are globally optimal algorithms that require knowledge of all node positions.

4.2.1 Greedy Furthest Radio. The *Greedy Furthest Radio* algorithm determines its furthest connected radio neighbor and commands it to rise. It is functionally identical to Algorithms 2. Instead of using acoustic connection data, we instead use radio connections in Algorithm 3. Just like *Greedy Furthest Acoustic*, this ignores node depth and may or may not require multiple acoustic-hops for the rise command to reach its destination. This resembles a locally-optimal algorithm that tries to minimize the number of participating nodes without accounting for depth. If all nodes are evenly spaced then this approximates a non-weighted minimum-hop approach. When nodes are spaced at or near the edge of acoustic or radio range is when this algorithm performs the worst; it becomes another "all-rise" scenario. The best case is when nodes are clustered together allowing many to be

skipped over, and the furthest neighbor is also the shallowest.

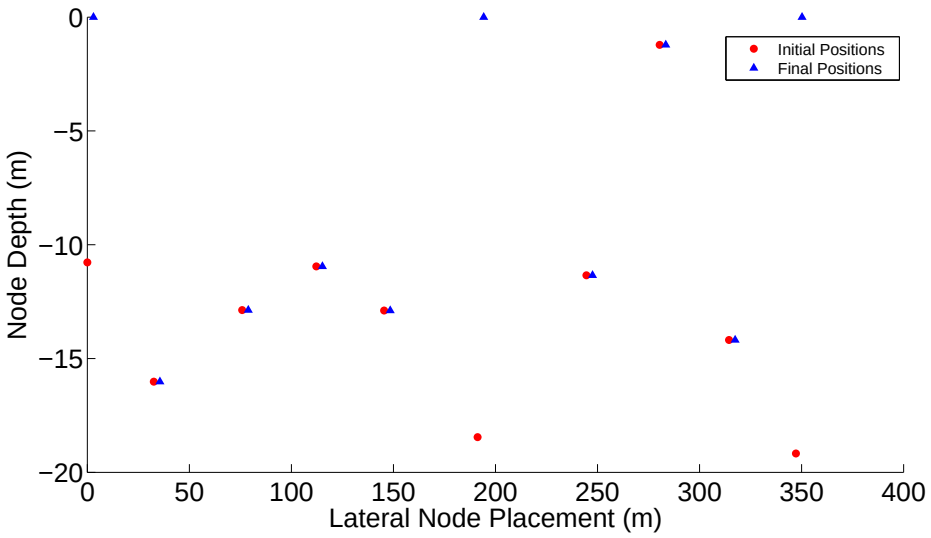

Figure 6: Graph of Start and End positions for Greedy Furthest Radio

Figure 6 shows initial and final positions for the Greedy Furthest Radio algorithm, with performance data presented in Table 7. This is the first radio-centric algorithm. The first thing to notice is the drastic change in distance travelled. Acoustic-centric algorithms are essentially forced in to using additional steps. In the case of acoustic range being half of radio range, which is default in the simulation, we would naturally require acoustic-centric algorithms to use approximately twice as many nodes to pass the intended message. A radio-based algorithm would then be expected to use roughly half the travel distance of an acoustic algorithm. Chapter 5 explores the effects of changing ranges in Sections 5.1.3 & 5.1.4. Comparing to Table 5, we are performing at or above the levels seen in *Greedy Furthest Acoustic*.

As we would expect, by taking a basic approach and providing another layer of information, algorithm performance is increased. The power requirement is less than 4.1.1.

This is solely due to the decreased number of nodes participating in routing. Fewer nodes are required to participate, so less power is used in moving.

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	2.9171	2.8147	2.6868	2.6114
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy(J)	45.00	41.48	40.19	39.74
Avg. Time (μs)	62.4871	63.4892	63.1409	63.0303

Table 7: Sample Statistics for Greedy Furthest Radio

4.2.2 Greedy Shallowest Radio. Similar to Greedy Shallowest Acoustic, this algorithm identifies the shallowest of its radio neighbors, which may or may not also be an acoustic neighbor. This approach is identical to Algorithm 4; the relationship between the *Greedy Furthest* algorithms is the same as the relationship between the *Greedy Shallowest* algorithms. All of the best and worse cases are the same, *mutatis mutandi* for radio communication. As for other algorithms, Figure 7 shows positions before and after this algorithm is simulated.

Table 8 contains performance values for the Greedy Shallowest Radio algorithm.

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	2.4561	2.0758	1.9844	1.9707
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy(J)	36.03	31.72	30.36	30.07
Avg. Time (μs)	62.1548	62.7176	62.3132	62.2241

Table 8: Sample Statistics for Greedy Shallowest Radio

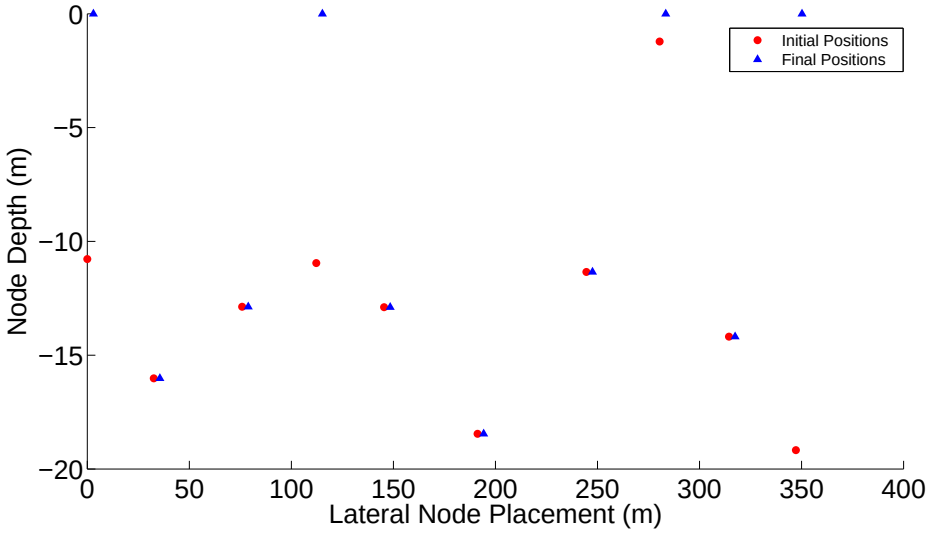

Figure 7: Graph of Start and End positions for Greedy Shallowest Radio

In comparison to 4.1.2 it is again made clear that with the increase in information, performance is increased. Just as when 4.2.1 was found to perform at over twice the levels of 4.1.1, so to does *Greedy Shallowest Radio* perform over twice as well as 4.1.2. Furthermore, as stated previously, *Greedy Shallowest Radio* out-performs *Greedy Furthest Radio*. The relationship follows exactly the same as for their acoustic counterparts.

4.2.3 Greedy Look-Ahead. This is the first algorithm that requires the addition of neighbor's-neighbors positions. Greedy Look-Ahead uses connection information spanning out to the furthest radio neighbor's furthest neighbor, and is described in Algorithm 7.

It then calculates the optimal next step using Dijkstra's algorithm with link-costs being weighted by node depth. The only requirements placed on the system are: no-backwards traversals (messages always advance) and the path chosen must have a length greater than two (not including the sender). The path length requirement is a safe-guard against becoming a purely shallowest-neighbor approach. The minimum weight path is selected, and

Algorithm 6 Overview of Greedy Look-Ahead

```
loop
    receive radio packet P
    if P.Destination ≡ Self then
        exit
    else
        queue P for delayed-transmission
        Connections ← connection data inlcuding furthest neighbor's furthest neighbor
        paths ← dijkstra(Positions, Connections, Self)          ▷ See Algorithm 7
        delete all paths from paths of length less than 2
        select lowest-cost path
        send rise command to next-hop
    end if
end loop
```

serves as an approximation of a globally optimal route.

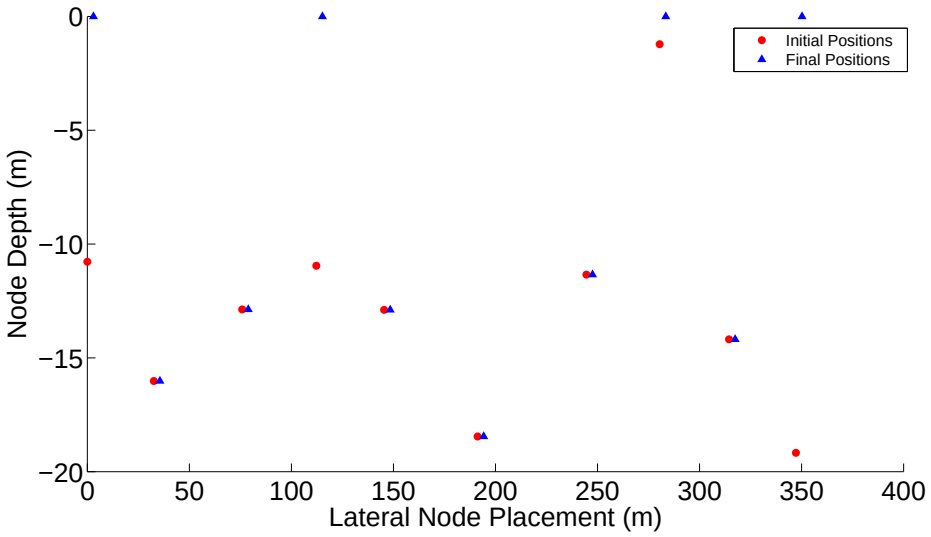

Figure 8: Graph of Start and End positions for Greedy Look-Ahead

The best case scenario is the route *Min-Hop Shallowest* (4.2.6) would take, and the worst case matches the worst case of Greedy Shallowest Radio. The average case does not perform as well as the former, and out-performs the latter. Figure 8 shows initial and final positions for Greedy Look-Ahead. Table 9 presents statistical data.

Algorithm 7 Finding locally-optimal, forward-moving paths; excerpt from MATLAB code

```
function id = greedy_look_ahead_algorithm(self_ID,dst_ID,Positions,Connections)
    immediateNeighbors = [];
    numNodes = length(Positions(:,1));
    distances = squareform(pdist(Positions(:,1)));

    for i=1:numNodes
        if Connections(self_ID,i) &&...
        (distances(i,dst_ID) < distances(self_ID,dst_ID))
            immediateNeighbors = [immediateNeighbors i];
        end
    end

    farthestNeighbor = immediateNeighbors(end);
    if farthestNeighbor == dst_ID
        cost = 0;
        id = dst_ID;
        return
    end

    immediateNeighbors = [];
    for i=1:numNodes
        if Connections(farthestNeighbor,i) &&...
        (distances(i,dst_ID) < distances(farthestNeighbor,dst_ID))
            immediateNeighbors = [immediateNeighbors i];
        end
    end

    farthestNeighbor = immediateNeighbors(end);

    relative_connections = Connections(self_ID:farthestNeighbor,self_ID:farthestNeighbor);
    costs = +relative_connections;
    for i=1:(farthestNeighbor - self_ID + 1)
        for j=1:(farthestNeighbor - self_ID + 1)
            if j < i
                relative_connections(i,j) = false;
                costs(i,j) = 0;
            else
                costs(i,j) = costs(i,j)*Positions((j + self_ID - 1),3);
            end
        end
    end

    costs = abs(costs);

    [costs,paths] = dijkstra(relative_connections,costs,1);

    costs(1) = [];
    paths(1) = [];

    for i=1:length(paths)
        if length(paths{i}) < 3 % require all paths to be [self next next-next]
            paths{i} = [];
        end
    end

    for i=length(paths):-1:1
        if isempty(paths{i})
            costs(i) = [];
            paths(i) = [];
        end
    end

    [~,id] = min(costs);

    id = paths{id};
    id = id(2) + self_ID - 1;
end
```

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	2.2097	1.8510	1.7401	1.6957
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy(J)	32.57	27.86	26.27	25.97
Avg. Time (μs)	79.3019	74.5612	68.6739	63.0941

Table 9: Sample Statistics for Greedy Look-Ahead

Greedy Look-Ahead observes a slight performance increase over 4.2.2. The improvement in distance comes at the expense of increased computational requirements. This algorithm relies on planning; plan creation will require the use of the higher power processor, as this method uses Dijkstra's algorithm to determine all paths that will have two additional hops, and selects the next hop of the lowest-cost path. As before, by adding in additional information we improve distance performance. With this particular method, only computational complexity increases, without increasing message passing.

4.2.4 Greedy Look-Back. *Greedy Look-Back* could be described as a time saving version of *Greedy Look-Ahead*. This algorithm sends a rise command to the *furthest* radio neighbor. Upon receipt of a rise command, the receiving node will check to see if it was the shallowest neighbor of the sending node. If so, it will rise and the algorithm continues forward. If not, it will invoke the look-ahead algorithm on behalf of the original sender, determine the appropriate node, and send a "forced-rise" command. This process is summarized in Algorithm 8. The forced-rise can not be further regressed, the receiving node is already part of the locally optimal path. The best and worst cases are the same as for Look-Ahead. The average case is similar to Greedy Look-Ahead, but a little slower and more power consuming. This average case is confirmed in Table 10. All average distances are identical to the distances in Table 9, which speaks more for the likelihood of the furthest

Algorithm 8 Overview of Greedy Look-Back

```
loop
    receive rise command P
    if P.Destination ≠ Self then
        forward P
    else
        if Self is shallowest neighbor of P then                ▷ See Algorithm 9
            rise
        else
            run Algorithm 6 for P.source
            send forced-rise command to next-hop
        end if
    end if
end loop
```

Algorithm 9 Finding locally-optimal, forward-moving paths; excerpt from MATLAB code

```
function id = greedy_look_back_algorithm(self_ID, dst_ID, requester_ID, Positions, Connections)
    immediateNeighbors = [];

    if dst_ID == self_ID
        id = self_ID;
        return
    end

    for i=(requester_ID+1):self_ID
        if Connections(self_ID,i) && (Positions(i,3) > Positions(self_ID,3))
            immediateNeighbors = [immediateNeighbors i];
        end
    end

    if isempty(immediateNeighbors)
        id = self_ID;
        return
    end
    id = greedy_look_ahead_algorithm(requester_ID, dst_ID, Positions, Connections);
end
```

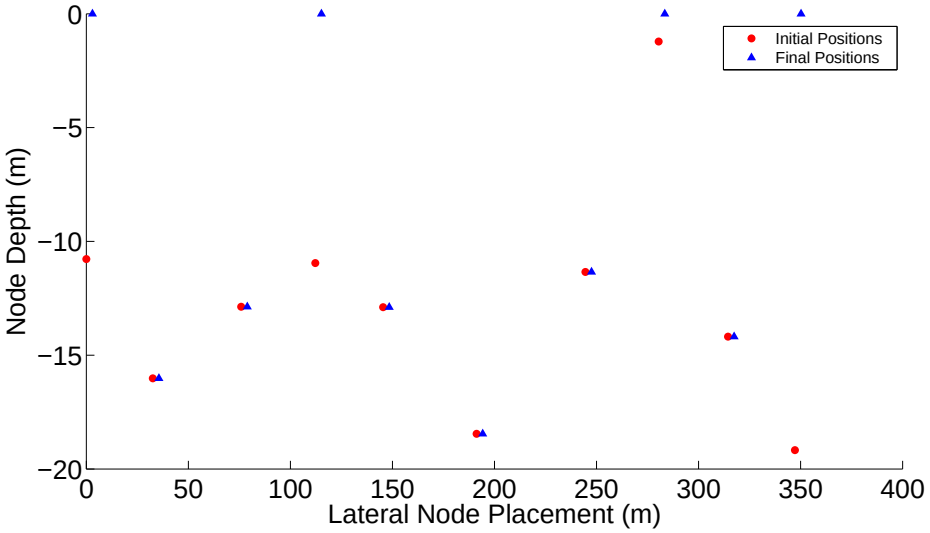

Figure 9: Graph of Start and End positions for Greedy Look-Back

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	2.2097	1.8510	1.7401	1.6957
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy(J)	32.57	27.86	26.27	25.97
Avg. Time (μs)	64.9102	67.6236	66.8615	66.5082

Table 10: Sample Statistics for Greedy Look-Back

neighbor being the shallowest than anything else.

4.2.5 Min-Hop Furthest. This is a globally optimal algorithm that does not take node depth into account. Using global position information, Dijkstra's algorithm is run with uniform costs applied to all network links. The optimal algorithms change behavior from Algorithm 2, as shown in Algorithm 10. The best case scenario is an improvement over *Greedy Furthest Radio*. The worst case occurs when the shortest possible path includes nodes deeper than would be selected by Greedy Furthest. When this happens, we see a large increase in energy consumption. It is not a common case, but it is worth being aware of. The average case is on par with *Greedy Furthest Radio*. Being a centralized

Algorithm 10 Optimal algorithm sequence of events

```
dijkstra(Positions, Connections, Node 1, Node N)
Send rise messages to all nodes on path                    ▷ See Algorithm 11
loop
    P ← received message
    if P.Destination ≡ Self then
        if Radio Message then
            exit
        else
            rise
        end if
    else
        forward message
    end if
end loop
```

Algorithm 11 Finding globally optimal paths, weighted and non-weighted; MATLAB code

```
function path = centralized_shortest_path_algorithm(self_ID, dst_ID,...
    Connectivity, Node_Pos, Weighted_or_not)

    NodeCount = size(Connectivity);
    NodeCount = NodeCount(1);

    if Weighted_or_not
        Weights = -Connectivity;
        for i=1:NodeCount
            for j=1:NodeCount
                Weights(i,j) = Weights(i,j)*Node_Pos(j,3);
            end
        end
    else
        Weights = +Connectivity;
    end

    for i=1:NodeCount
        Weights(i,i) = inf;
    end

    [~,path] = dijkstra(Connectivity,Weights,self_ID,dst_ID);
end
```

algorithm has its consequences. Run-time increased over 4.2.1, with little effect on energy usage. The reason for the increase in run time is computational complexity (Dijkstra's algorithm). Every single message generated needed to be transmitted from the first node, to its destination. This includes rise packets. The power cost is elevated so greatly because the closer a node is to the origin, the more it is required to route. This is the same sort of issue that would be seen in a standard sensor network where all nodes share a common sink. Being close to the origin results in an uneven load on the system, which could be mitigated by distributing the algorithm or by including next-hops as data in a custom rise

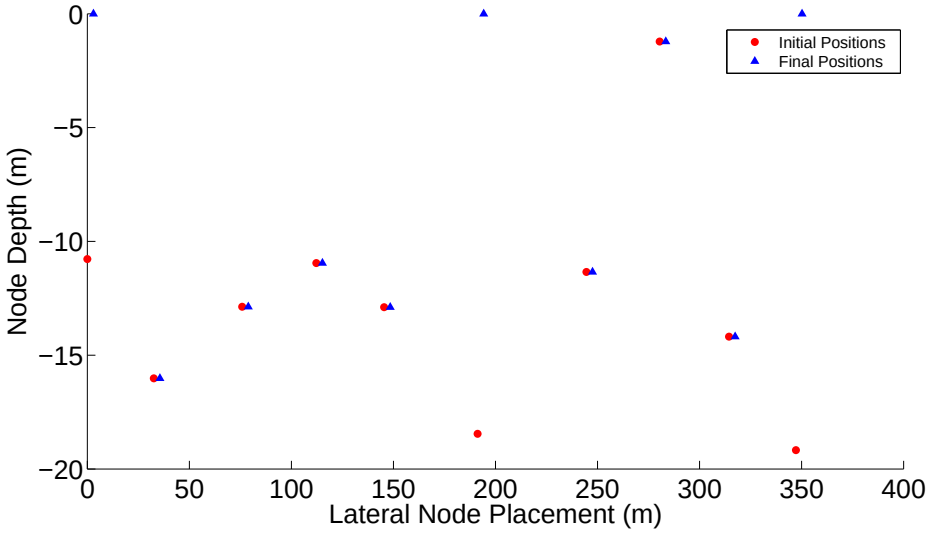

Figure 10: Graph of Start and End positions for Min-Hop Furthest

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	3.0701	2.7676	2.6652	2.6415
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy(J)	44.44	41.58	40.66	39.95
Avg. Time (μs)	65.6124	74.9882	80.8565	87.6294

Table 11: Sample Statistics for Min-Hop Furthest

command.

4.2.6 Min-Hop Shallowest. Using the global connectivity matrix and weighing links in the matrix by node depth, Dijkstra's algorithm determines the optimal *weighted* path from start to finish, consistently guaranteeing the least node movement. This follows the exact same approach as in Algorithm 10; when Algorithm 11 executes, the `Weighted_or_not` flag is true. Similar to 4.2.5, there is an excessive energy burden based on the algorithm being centralized. Ignoring that component, we should turn our attention to distance travelled and time, as shown in Table 12. The average distances travelled are lower than any other algorithm. The closest approximations come from the two planning algorithms (4.2.3 and

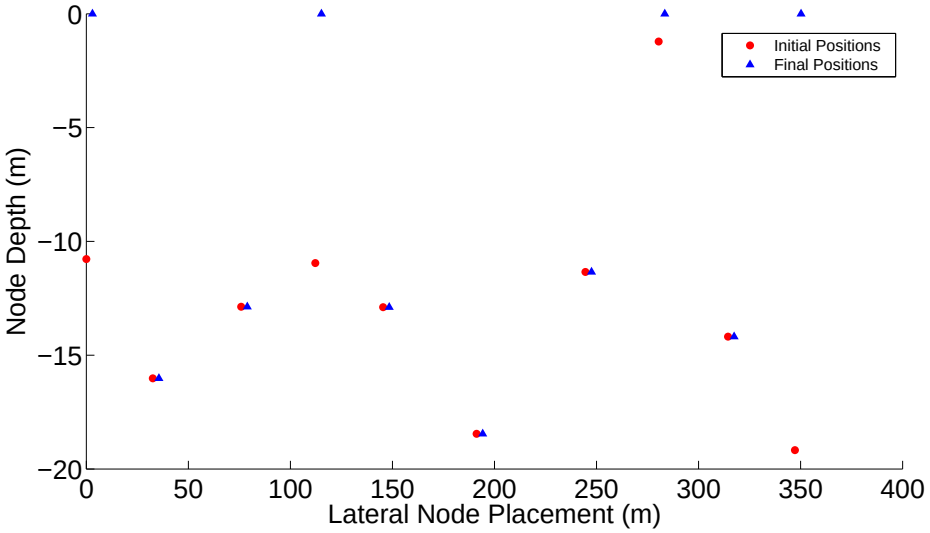

Figure 11: Graph of Start and End positions for Min-Hop Shallowest

	Number of Nodes			
	25	**50**	**75**	**100**
Avg. Distance (m)	2.1485	1.8068	1.6963	1.6591
Avg. Depth (m)	10.1220	9.9952	9.9465	10.0011
Avg. Energy(J)	31.89	27.21	25.66	25.32
Avg. Time (μs)	85.9095	86.8690	85.1852	84.1778

Table 12: Sample Statistics for Min-Hop Shallowest

4.2.4), which each observe a much lower average energy cost. If all nodes were allowed global information and the optimal algorithm were spread across nodes, then optimal paths could be generated at the same basic cost as *Greedy Look-Ahead*.

If only information on immediate neighbors can be known, the clear choice must be *Greedy Shallowest Radio*. If we allow for information on our neighbor's neighbors, the decision between *Greedy Shallowest Radio* and *Greedy Look-Ahead* depends on the cost of computation. The former is a close approximation of the latter, which is a close approximation of the ideal. For sake of simplicity, *Greedy Shallowest Radio* likely remains the strongest candidate.

Chapter 5: Experiments and Results

In this chapter we discuss the set of experiments we conducted and their results. We chose to conduct a total of four experiments. The first is using default simulation settings and across all eight algorithms discussed in Chapter 4. The second explores the effect of changing how nodes are placed; first by having a fixed maximum distance and moving the minimum towards it, and then by using a fixed minimum distance and bringing the maximum down to it. The third uses a fixed radio range, and increases acoustic range up to the radio capabilities. Lastly, we use a fixed acoustic range, and demonstrate the effect changes in radio ranges have. We end this chapter with a brief discussion of the results and draw conclusions on which algorithms are optimal, and how this answer can change.

5.1 Experiments

The simulator requires several settings to be defined beyond the number of nodes and length of message queues. These settings include minimum and maximum lateral node placement (by default: 30 to 60 meters apart on the x-axis), minimum and maximum node depth (ranging from -20m to 0m depth), and the range of the acoustic and radio modems. It will be noted when these values are modified, the full set of features and default values can be seen in Appendix A.1.1.

5.1.1 Basic System, All Algorithms. All default values are preserved in this experiment, as described in Table 13. These settings ensure that each node has at least one

System Default Values	
Acoustic Range (m)	100
Radio Range (m)	200
Minimum X-axis Spacing (m)	30
Maximum X-axis Spacing (m)	60
Minimum Y-axis Spacing (m)	0
Maximum Y-axis Spacing (m)	0
Maximum Depth (m)	20

Table 13: Basic settings that will determine simulator behavior.

acoustic neighbor closer to the destination than itself and, depending on a node's placement in the network, at least three radio neighbors closer to the destination than itself. Eight algorithms are run on network sizes from ten to 100 nodes, in increments of ten, across 500 unique topologies. Figure 12 shows the average amount of motion any given node could expect to move, at a given network size. The error bars on the plots represent standard deviation, and are only shown for algorithms that are significantly different from each other.

The upper plot contains data on the acoustic-centric algorithms, with *Greedy Shallowest Acoustic* showing estimates on error. The lower plot contains results on all but two radio-centric algorithms (excluded are *Greedy Look-Back* and *Min-Hop Furthest*). Both *Greedy Shallowest Radio* and *Min-Hop Shallowest* show error margins. We choose to separate acoustic and radio algorithms because they are effectively on different scales. The acoustic algorithms require more motion, and the standard deviation for the acoustic algorithms is considerably larger than for the radio algorithms.

Four algorithms are selected to have their energy use shown in Figure 13. They are *Greedy Shallowest Acoustic*, *Greedy Shallowest Radio*, *Greedy Look-Ahead*, and *Min-Hop Shallowest*. We chose to present an acoustic algorithm to show the vast discrepancy between acoustic- and radio-based algorithms, and the radio algorithms shown will continue

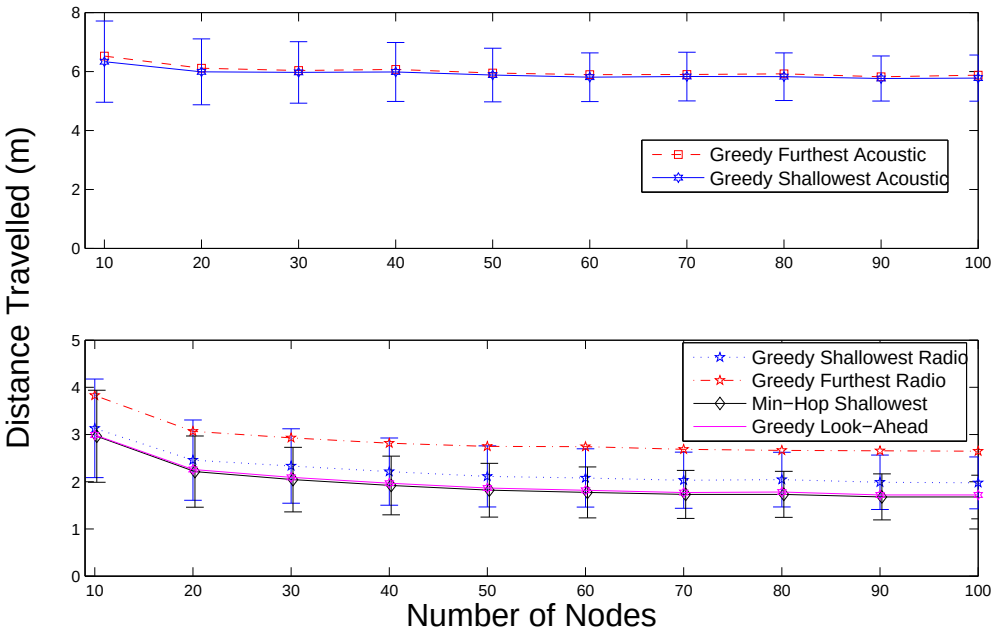

Figure 12: Graph of Start and End positions for Greedy Furthest Acoustic

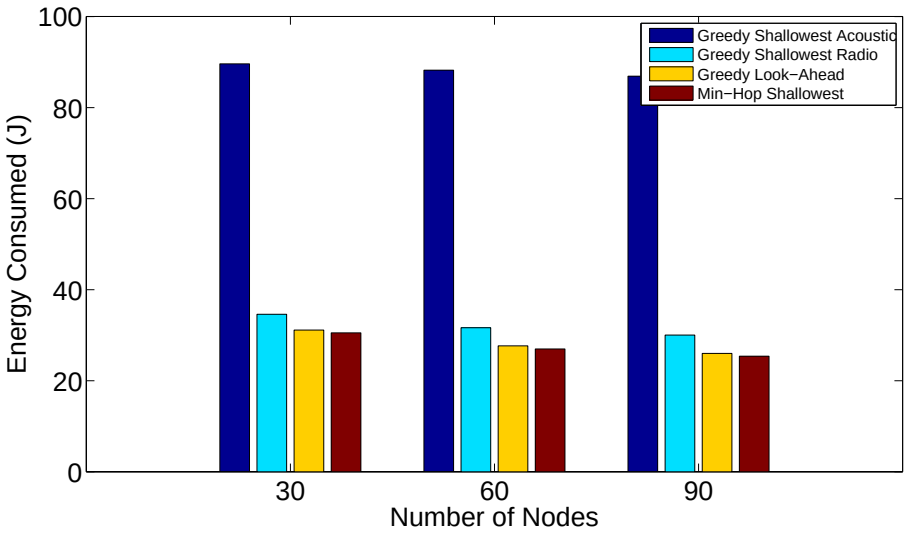

Figure 13: Graph of Energy Consumed by Four Algorithms

	System Default Values	Experiment Values
Acoustic Range	100	100
Radio Range	200	200
Minimum X-axis Spacing (m)	30	**10–60**
Maximum X-Axis Spacing (m)	60	**80**
Maximum Depth (m)	20	20

Table 14: Settings for the first phase: increasing minimum node spacing

	System Default Values	Experiment Values
Acoustic Range	100	100
Radio Range	200	200
Minimum X-axis Spacing (m)	30	**10**
Maximum X-Axis Spacing (m)	60	**70–40**
Maximum Depth (m)	20	20

Table 15: Settings for the second phase: reducing maximum node spacing

to be used as examples for the duration of this thesis. Power savings between radio algorithms is much smaller, but that difference becomes more noticeable over long periods as well. A point of interest with Figure 13 is that as the number of nodes rises, power consumed by the two most-optimal algorithms shown (*Look-Ahead* and *Min-Hop*) stays very close. Recall that the globally optimal algorithms are centralized, and furthermore they use the first node as the origin point for *all packets*. Due to this implementation decision, the optimal algorithm has an artificially inflated power requirement. It is readily apparent from Figures 12 and 13 that in a typical situation, a radio algorithm is a better choice.

5.1.2 Varying Node Placement. This experiment is in two phases. The first is with a fixed maximum distance and increasing minimum distance, and the second uses a fixed minimum while lowering the maximum. The settings for the first phase are described in Table 14.

In Figure 14 we show the average distance travelled for a node using four algorithms. The lower plot describes when the maximum and minimum are within 20m of each other,

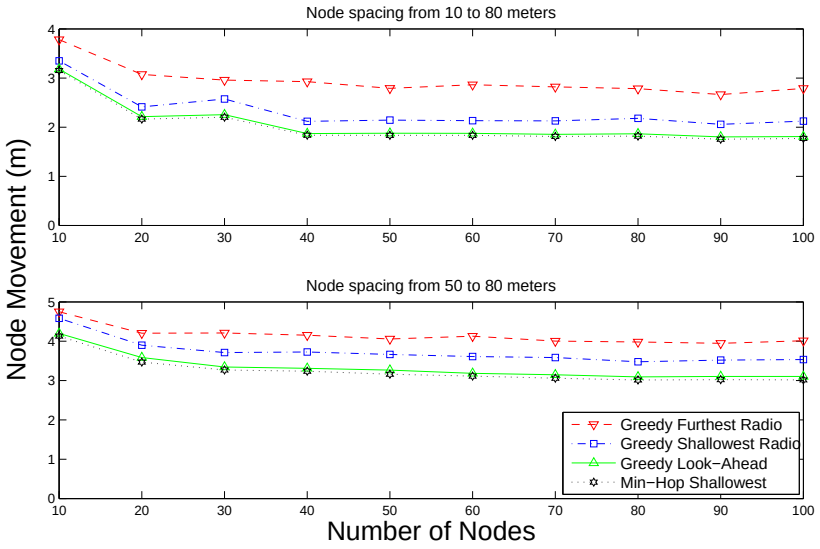

Figure 14: The Effect Separating Nodes Further on Distance

and close to the end of acoustic range. Figure 15 shows the effect separation has on a network of 50 nodes. We observe from Figure 14 that in general, all algorithms fare worse with more separation. This impact is highlighted by the upward trend in Figure 15. As space between nodes is increased, required movement increases. Figure 16 details the power requirements for the same four algorithms. Obvious at a glance is that power consumption increases dramatically as nodes are placed further apart. As distance-travelled increases, power increases. We observe that distance increases with separation; greater separation leads to fewer neighbors to choose from, which leads to more nodes participating in routing, which then requires greater distance travelled. This trend is highlighted in Figure 17 which demonstrates the linear increase in power usage.

The second phase of this experiment is described in Table 15. Instead of increasing the minimum separation, we vary the maximum separation. Between Figures 18 and 19 it is made clear that moving nodes closer together improves their performance. Data is only shown for the wide range of 50m of separation from the default minimum (upper plot)

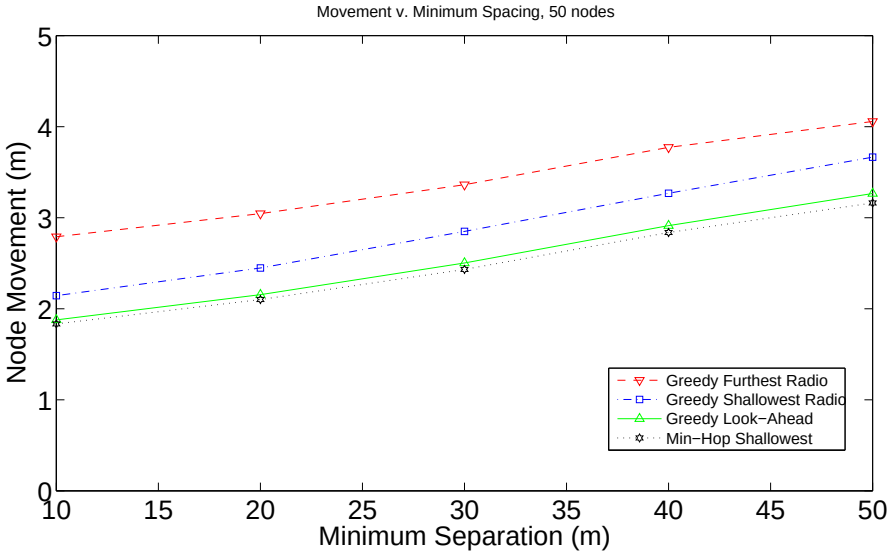

Figure 15: The Effect on 50 Nodes Being Separated Further

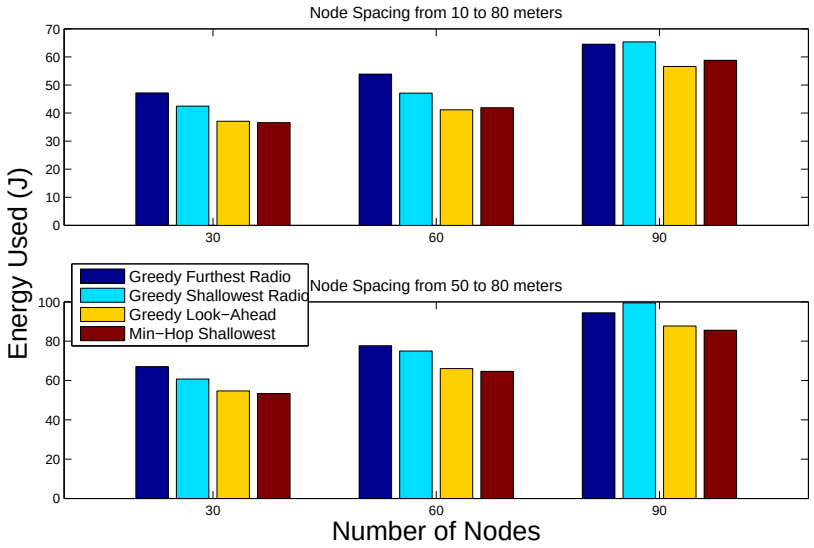

Figure 16: The Effect Separating Nodes Further on Energy Use

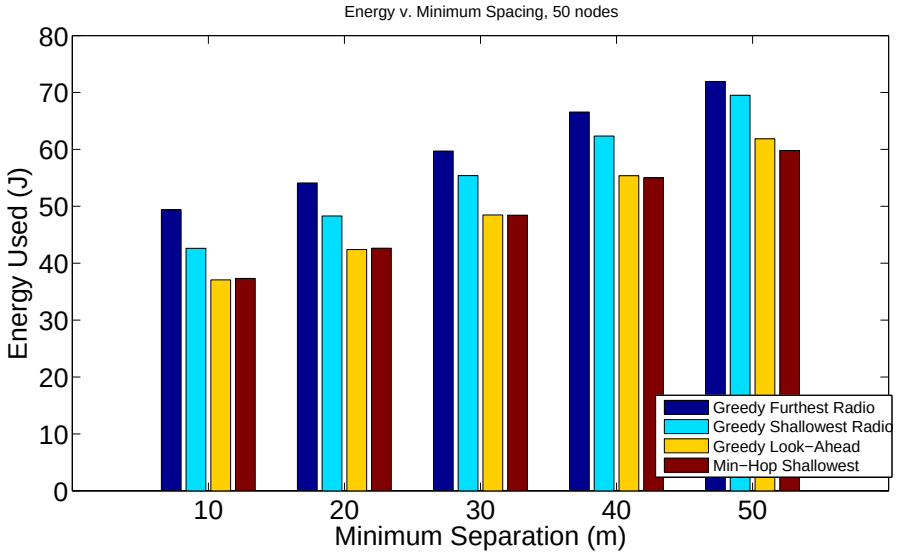

Figure 17: The Effect on Power Consumption Increased Separation has on 50 Nodes

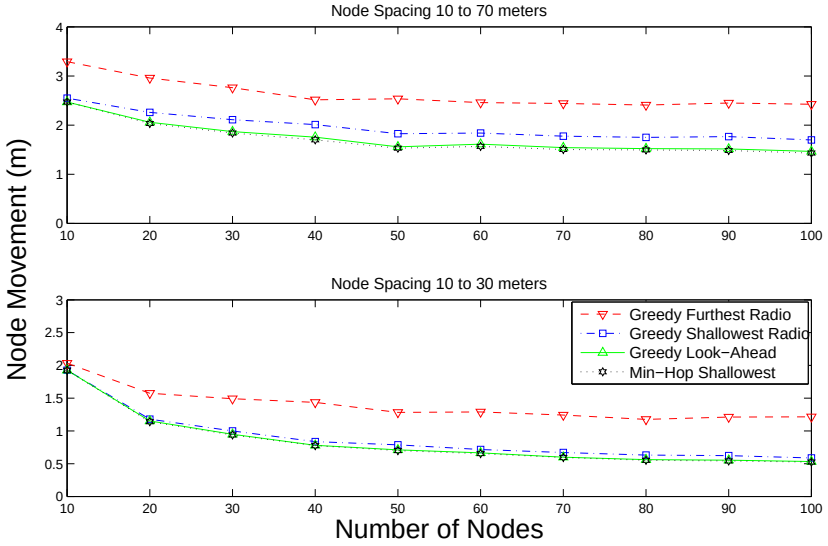

Figure 18: The Effect Placing Nodes Closer Together on Distance

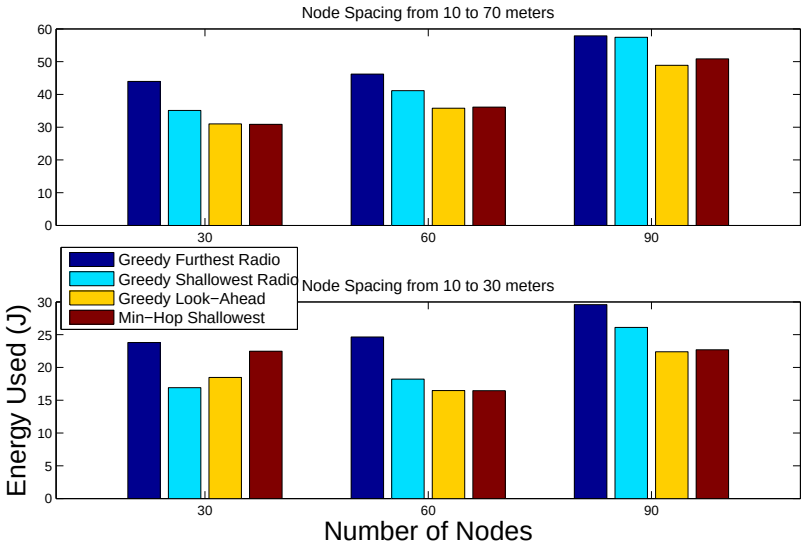

Figure 19: The Effect of Placing Nodes Closer Together on Energy Use

and even smaller range of 20m of separation (lower plot). As in the case of moving nodes further apart, we show in Figures 20 and 21 that nodes placed closely together move less and require less energy. Similar to increasing minimum spacing, as maximum spacing decreases the number of nodes required to move drops, and so there is a general performance boost.

We can safely state that, communication abilities aside, this system is more power efficient in close ranges. In this case it is a natural result of the movement capabilities. The cost of moving along the z-axis is very high, so any reduction in movement is a performance boost. Moving nodes closer together requires less lateral travel for messages, which require fewer nodes to participate in message routing. The difference in distances are shown between Figures 14 and 18, and as distance is essentially a proxy for power, it is natural that Figures 16 and 19 would follow their general trend. If power cost of movement were ignored, the next largest contributor to consumption would be acoustic communication. The closer nodes are, the fewer acoustic messages would be necessary for even the

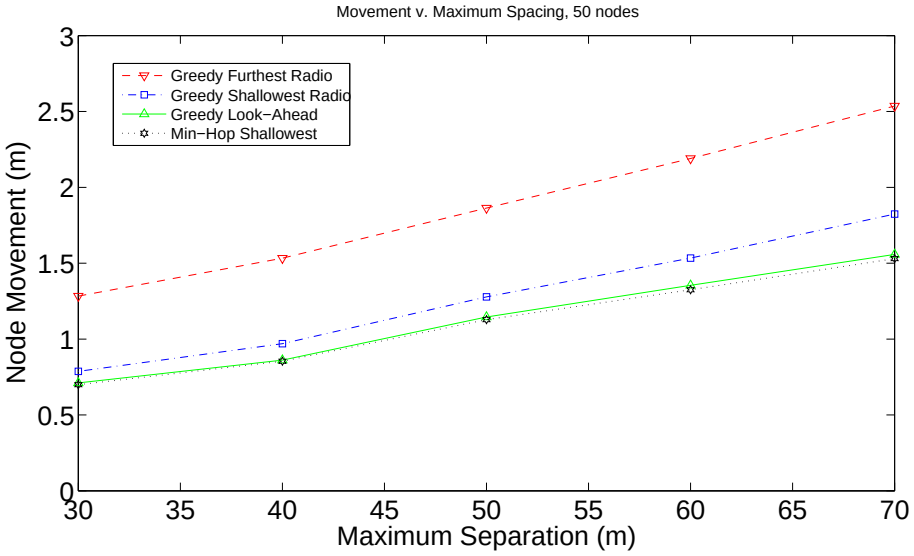

Figure 20: The Effect Placing Nodes Closer Together on Distance

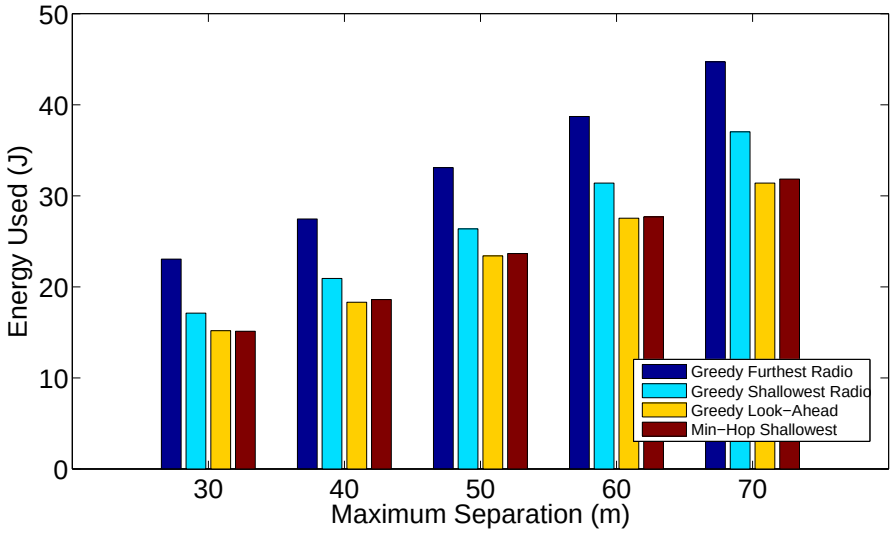

Figure 21: The Effect of Placing Nodes Closer Together on Energy Use

	System Default Values	Experiment Values
Acoustic Range	100	**120–200**
Radio Range	200	200
Minimum X-axis Spacing (m)	30	30
Maximum X-Axis Spacing (m)	60	60
Maximum Depth (m)	20	20

Table 16: Experiment settings: increasing acoustic range

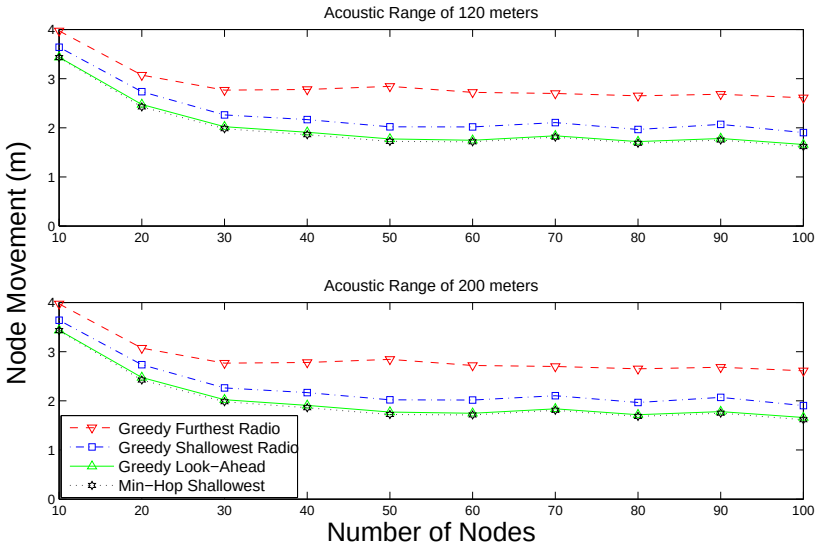

Figure 22: Effects of Increasing Acoustic Range on Distance Travelled

most involved algorithms (e.g. *Greedy Look Back*).

5.1.3 Varying Acoustic Range, Fixed Radio Range. In this experiment we use default position and radio settings, and vary acoustic range. Acoustic range is varied 120m to 200m, as shown in Table 16, and range moves in 20m increments. We find an interesting result in Figure 22. Acoustic range does not impact distance travelled for radio algorithms. It is not without any effect, however. Observing Figure 23 clearly shows a change in power consumption. The lower plot represents increased range, while the upper is closer to standard range. We observe that distance travelled with radio algorithms is not affected by changes in acoustic range. Further, it should be noted that power requirements actually

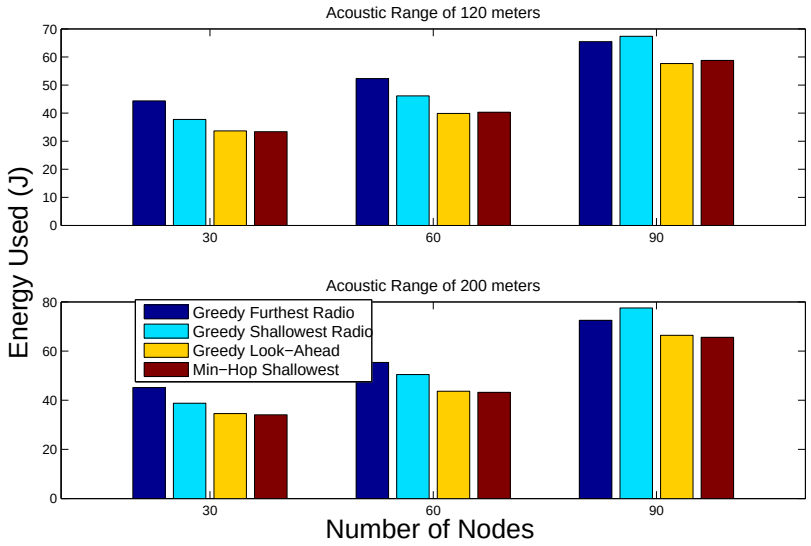

Figure 23: Effects of Increasing Acoustic Range on Energy Used

	System Default Values	Experiment Values
Acoustic Range	100	100
Radio Range	200	**150–450**
Minimum X-axis Spacing (m)	30	30
Maximum X-Axis Spacing (m)	60	60
Maximum Depth (m)	20	20

Table 17: Varying radio range, 150 meters to 350 meters.

slightly increased, as shown in Figures 23 and 24. This is due to an increase in acoustic message forwarding that occurs with increased range. It is a very small contribution when compared to node motion.

5.1.4 Varying Radio Range, Fixed Acoustic Range. All default settings remain constant, except radio range. Radio range was allowed to go below its default value but above acoustic range, as shown in Table 17, and was allowed to go an additional hundred meters beyond standard range. Radio range increased in 50m increments. Figure 25 shows the impact this has on distance, and Figure 26 shows this result on 50-node networks. Naturally, the change is quite dramatic. The further a node is able to reach, then necessarily

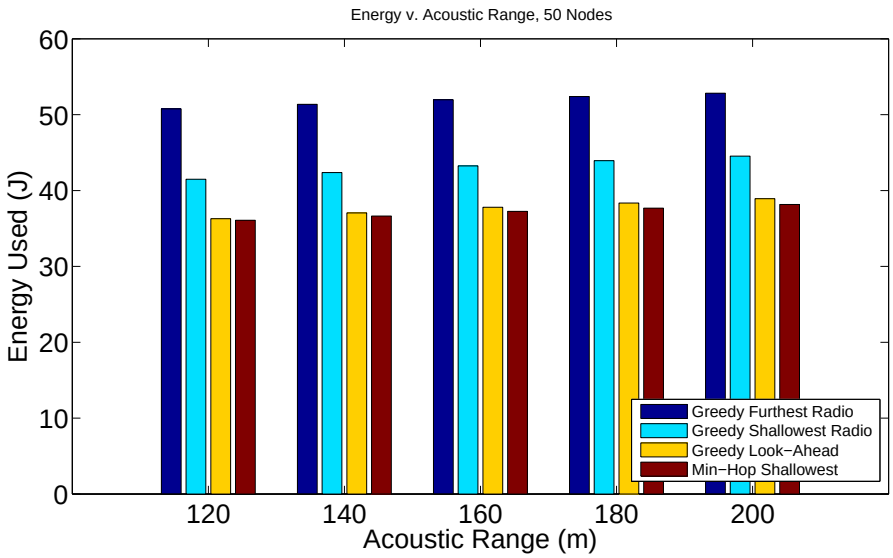

Figure 24: Effects of Increasing Acoustic Range on Energy Used, 50 nodes

fewer nodes must participate in rising and routing. As mentioned previously, power and movement follow the same general trends.

Figures 27 and 28 shows the impact this has on energy use. The pair of plots demonstrate that radio range is a major indicator of algorithm efficiency. As radio capabilities approach acoustic capabilities, performance degrades. In that same way, performance increases as capabilities are enhanced.

5.2 Discussion

We conducted a total of four experiments. In those experiments we explored the result of manipulating simulation defaults. We started by developing baseline data, seen in 5.1.1. We could observe that acoustic algorithms are less efficient than radio algorithms, and it is clearly shown that algorithms that account for more of the network fare better.

From the data, the ideal network has: equal acoustic and radio capabilities, these both have range several times longer than maximum node spacing, and all nodes are placed close

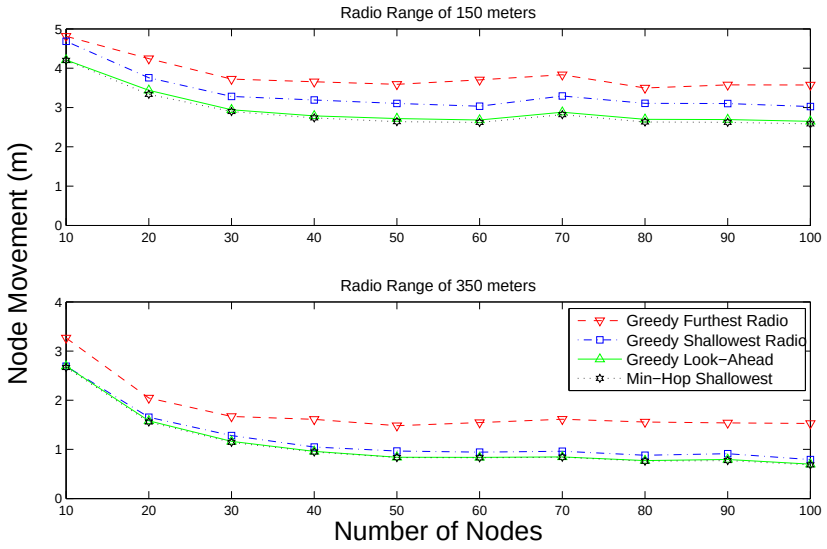

Figure 25: Effects of Changing Radio Range on Distance Travelled

together. It is not realistic to assume all of the features are possible together, or that if they were it would suit any given system. What is of more importance than strict ranges/displacements is the ratio of those to modem range. In a densely packed system, we could use a lower power acoustic modem, as acoustic range did not have a large impact on mobility and the power gain from bringing nodes closer together exceeded that of increasing acoustic range. For sparse systems where nodes are far apart, we need to invest in powerful radios to offset the acoustic burden. In any case it is important to keep record of neighbor positions, and it can be seen in all experiments that providing additional information to packet routing always yields a beneficial result.

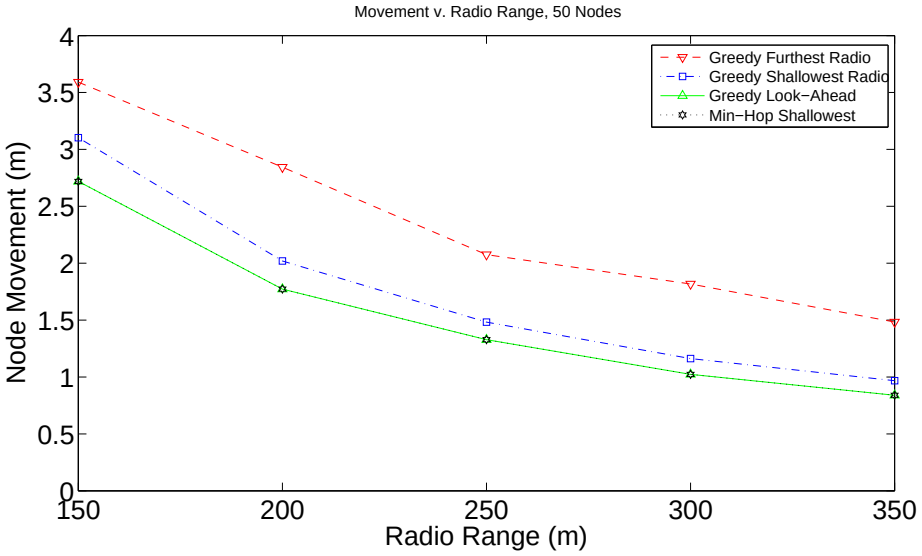

Figure 26: Effects of Changing Radio Range on Distance Travelled for 50-Node Networks

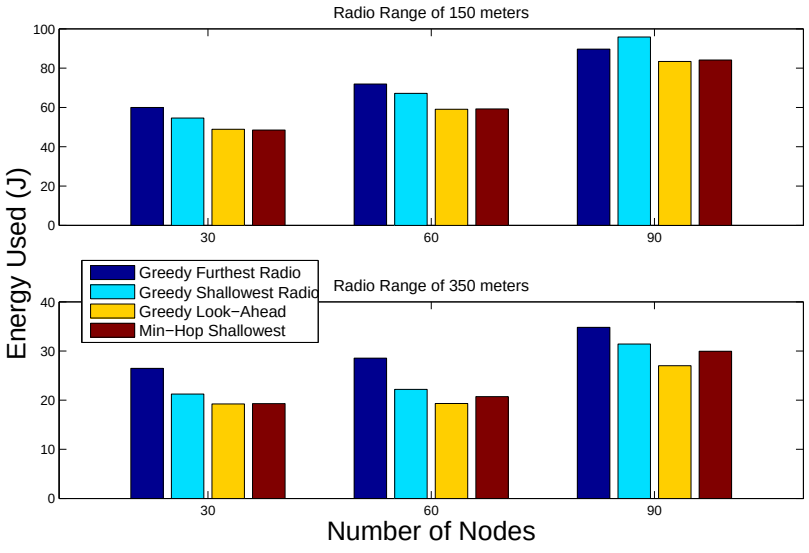

Figure 27: Effects of Changing Radio Range on Energy Used

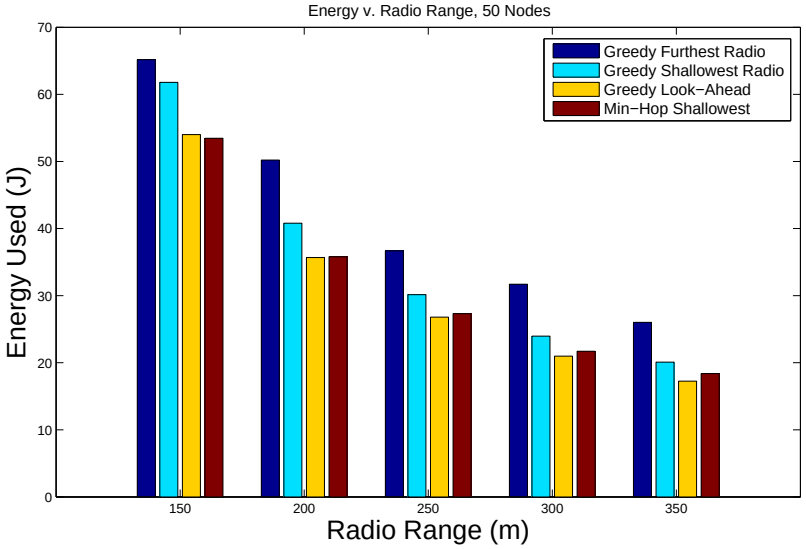

Figure 28: Effects of Changing Radio Range on Energy Used

Chapter 6: Conclusions

Networking and routing are problems that have been solved on land years ago, but due to the unique constraints of the underwater environment we must solve them again. This is a fledgeling field, so we assist in this endeavor by developing an analytical tool and describing who should participate in routing. To that end we developed a simulation environment in MATLAB, and implemented eight algorithms to make the participation decision. Between those algorithms and the behavior of the simulator, we provide suggested approaches to managing radio use. A large number of test cases were required to verify the accuracy as can be seen in chapter 5, but we are confident in our recommendations.

6.1 Summary In Brief

Using MATLAB we developed a set of scripts and functions that replicated the basic functionality of an AquaNode [1] in an idealized environment. We assume perfect communication, and we do not presently allow for a node to be temporarily or permanently disabled. Within this ideal system, where energy is consumed and tracked but not limited, we develop and test eight algorithms. Two of them were based on the acoustic capabilities, the other six based on radio capabilities. The acoustic-algorithms and first two radio-algorithms share the same decision process. They are ”furthest neighbor” and ”shallowest neighbor” approaches. We show that using the radio approach is more effective as long as the range of the radio is greater than acoustic range, and as their ranges come closer

together we see their performances converging.

The primary lesson learned is that radio-centric algorithms are generally more efficient, and the more information available allows for more economic motion and power consumption. If an agent only knows the positions of its neighbors, *Greedy Shallowest Radio* is the best choice. If processing power is available and we have information on our neighbors and their neighbors, *Greedy Look-Ahead* is the best choice, assuming we don't want a centralized approach. These two algorithms allow for a fully distributed approach using local or semi-local information only.

6.2 Contributions

We have two main contributions. The first is the creation of an aquatic simulator that allows for rapid prototyping and validation of algorithms. The second is the evaluation of basic schemes that can be readily implemented on a physical system, while providing a rough estimate of their efficacy.

As stated in Chapter 1, we developed an environment based on AquaNodes. The system is implemented in MATLAB, so development times are very short. This allowed us and can allow others to quickly create, test, and modify algorithms and simulator extensions. We demonstrate this through the algorithms we implemented. Each of them were implemented in an abstract fashion which allowed for proof of concept. The abstract implementation demonstrates how the real system should behave, and so provides a framework to progress in more physical versions.

6.3 Future Work

There are two primary ways the work of this thesis can be extended. The first is by extending the capabilities of the simulator itself. The second is through implementing the algorithms on a physical system.

Some ways the simulator could be expanded are: allowing node sleep or power loss, implementing the behavior of the AquaNode multiprocessor system (including variable power requirements), and implementing communication interference. Sleeping nodes and the multiprocessor system would allow for more realism in power consumption, and thus the trends we show in chapter 5 would be more assuredly held to, and any power saving efforts could be tested in a time-accelerated environment. Interference takes multiple forms. The first is by varying the success rate of communication to serve as a proxy for general conditions. Data for this is available in [1]. The simulator could be further expanded to having environmental settings, such as the existence and persistence of thermoclines. A truly accurate system would track the *acoustic power* of all relevant transmissions and determine simple noise-interference from other nodes "talking" or messages reflecting off of surfaces.

The goal of this thesis has been to develop a simulation environment and a sample set of algorithms for evaluation. It is our sincere hope that this tool will be of use to researchers working on wireless underwater networks. There are many new and exciting approaches to the problems aquatic systems face, and we hope we've made the process of exploring these approaches a little easier. It is critical we properly develop and leverage aquatic technology so we can understand and make use of what the ocean provides.

REFERENCES

[1] C. Detweiler, *Decentralized Sensor Placement and Mobile Localization on an Underwater Sensor Network with Depth Adjustment Capabilities*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, July 2010.

[2] C. Detweiler, M. Doniec, I. Vasilescu, E. Basha, and D. Rus, "Autonomous depth adjustment for underwater sensor networks," in *Proc. of the Intl Workshop on Under-Water Networks (WUWNet)*, p. 12:112:4, 2010.

[3] C. Detweiler, M. Doniec, I. Vasilescu, and D. Rus, "Autonomous depth adjustment for underwater sensor networks: Design and applications," *IEEE Transactions on Mechatronics*, vol. 17, no. 1, pp. 16–24, 2012.

[4] I. Vasilescu, *Using Light Underwater: Devices, Algorithms and Systems for Maritime Persistent Surveillance*. PhD thesis, MIT, 2009.

[5] M. O'Rourke, E. Basha, and C. Detweiler, "Multi-modal communications in underwater sensor networks using depth adjustment," (Los Angeles, CA, USA), 2012.

[6] C. Chen and J. Ma, "Mobile enabled large scale wireless sensor networks," in *Advanced Communication Technology, 2006. ICACT 2006. The 8th International Conference*, vol. 1, p. 333338, 2006.

[7] S. Krco, D. Cleary, and D. Parker, "P2P mobile sensor networks," in *System Sciences,*

2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on, p. 324c, 2005.

[8] I. F. Akyildiz, D. Pompili, and T. Melodia, "State-of-the-art in protocol research for underwater acoustic sensor networks," in *Proc. of the 1st WUWNet*, (Los Angeles, CA, USA), p. 716, 2006.

[9] J. Rice, "SeaWeb acoustic communication and navigation networks," in *Proceedings of the International Conference on Underwater Acoustic Measurements: Technologies and Results*, 2005.

[10] J.-H. Cui, J. Kong, M. Gerla, and S. Zhou, "The challenges of building mobile underwater wireless networks for aquatic applications," *Network, IEEE*, vol. 20, p. 1218, June 2006.

[11] Z. Zhou, H. Yan, S. Ibrahim, J.-H. Cui, Z. Shi, and R. Ammar, "Enhancing underwater acoustic sensor networks using surface radios: Issues, challenges and solutions," in *Sensor Networks* (G. Ferrari, ed.), Signals and Communication Technology, p. 283307, Springer Berlin Heidelberg, 2009.

[12] B. Chen, P. C. Hickey, and D. Pompili, "Trajectory-aware communication solution for underwater gliders using WHOI micro-modems," in *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*, p. 19, 2010.

[13] N. Arad and Y. Shavitt, "Minimizing recovery state in geographic ad-hoc routing," in *Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, MobiHoc '06, (New York, NY, USA), p. 1324, ACM, 2006.

[14] G. Tan, M. Bertier, and A.-M. Kermarrec, "Visibility-graph-based shortest-path geographic routing in sensor networks," in *INFOCOM 2009, IEEE*, p. 17191727, Apr. 2009.

[15] Z.-Y. He and P. Jiang, "Design of wireless gateway based on ZigBee and GPRS technology," in *2009 International Conference on Computational Intelligence and Software Engineering*, (Wuhan, China), pp. 1–4, Dec. 2009.

[16] H.-j. He, Z.-q. Yue, and X.-j. Wang, "Design and realization of wireless sensor network gateway based on ZigBee and GPRS," in *2009 Second International Conference on Information and Computing Science*, (Manchester, England, UK), pp. 196–199, May 2009.

[17] S. Chumkamon, P. Tuvaphanthaphiphat, and P. Keeratiwintakorn, "The vertical handoff between GSM and zigbee networks for vehicular communication," in *Electrical Engineering/Electronics Computer Telecommunications and Information Technology (ECTI-CON), 2010 International Conference on*, p. 603606, 2010.

[18] S. Lin, S. xiangquan, and L. Ming, "A wireless network based on the combination of zigbee and GPRS," in *2008 IEEE International Conference on Networking, Sensing and Control*, (Sanya, China), pp. 267–270, Apr. 2008.

APPENDIX A: MATLAB CODE

A.1 Initializers

```
if ~exist('numberNodes','var')
  numberNodes = 100;
end

if ~exist('queueLength','var')
  queueLength = ceil(numberNodes/2);
end

if ~exist('mode','var')
  mode = 8;
end

if ~exist('ACOUSTIC_INDEX','var')
  ACOUSTIC_INDEX = 2;
end

if ~exist('ACOUSTIC_RANGE','var')
  ACOUSTIC_RANGE = 100;
end

if ~exist('ACOUSTIC_SEND_ENERGY','var')
  ACOUSTIC_SEND_ENERGY= 0.1136/1000;
end

if ~exist('ACOUSTIC_RECEIVE_ENERGY','var')
  % estimate until I have real values
  ACOUSTIC_RECEIVE_ENERGY = ACOUSTIC_SEND_ENERGY/1.8;
end % based of change in current from Detweiler thesis p.71, table 3.1

if ~exist('RADIO_RANGE','var')
  RADIO_RANGE = 200;
end

if ~exist('RADIO_INDEX','var')
  RADIO_INDEX = 1;
end
```

```
if ~exist('RADIO_SEND_ENERGY','var')
  RADIO_SEND_ENERGY = 0.00016/1000;
end

if ~exist('RADIO_RECEIVE_ENERGY','var')
  RADIO_RECEIVE_ENERGY = RADIO_SEND_ENERGY/3.5;
end % based of change in current from Detweiler thesis p.71, table 3.1

if ~exist('ACOUSTIC_SUCCESS','var')
  ACOUSTIC_SUCCESS = 1;
end

if ~exist('RADIO_SUCCESS','var')
  RADIO_SUCCESS = 1;
end

if ~exist('MIN_SEND_DELAY','var')
  MIN_SEND_DELAY =10;
end

if ~exist('PROCESSING_ENERGY','var')
  % 4*numberNodes*this == process power per iteration
  % below is 3.3*.059, V*I (Watts)
  PROCESSING_ENERGY = 3.3*(.059/1000)*(4*numberNodes);
  % this is the
end

if ~exist('WINCH_ENERGY','var')
  WINCH_ENERGY = 15; % 15J/meter, multiply by
end

% Parameters for topology init

MINIMUM_X = 30;
MINIMYM_Y = 0;
MINIMUM_Z = -20;

MAXIMUM_X = 60;
MAXIMUM_Y = 0;
% MAXIMUM_Z = 0;



% Create the initial topology matrix (n,3) where n=number of nodes
%
% Can either use an existing one that is passed via input or create a
% random one based on input constraints

Node_Positions = zeros(numberNodes,3);
```

```
if exist('Node_Positions','var')
  numberNodes = length(Node_Positions(:,1));
else
  Node_Positions = zeros(numberNodes,3);
  prev = [0 0 0];
  for i=1:numberNodes
    x = 0;
    y = 0;
    z = 0;
    while x < MINIMUM_X || x > MAXIMUM_X
      if i == 1
        x = 0;
        break;
      end
      x = rand(1)*MAXIMUM_X;
    end
    x = x + prev(1);
%     while y < MINIMYM_Y || y > MAXIMUM_Y
%       y = rand(1)*MAXIMUM_Y;
%     end
%     y = y + prev(2);
    y = 0;
    %
    % Z operates on negatives, so we go from max (assumed 0) to min
    % This means we go from a negative to zero. Just look at the code!
    z = rand(1)*MINIMUM_Z;
    %
    %
    %
    prev = [x y z];
    Node_Positions(i,:) = prev;
  end
end
clear i x y z prev
% clear MAXIMUM_X MAXIMUM_Y %MAXIMUM_Z
% clear MINIMUM_X MINIMYM_Y MINIMUM_Z



% Create connectivity matrix based on topology
%
<<<<<<< local
% Inputs: Node_Positions, numberNodex
=======
% Inputs: Node_Positions,ACOUSTIC_RANGE,RADIO_RANGE,RADIO_INDEX,ACOUSTIC_INDEX
>>>>>>> other
```

```
% Outputs: Connectivity_Matrix

% Create empty matrix
Connectivity_Matrix = zeros(numberNodes,numberNodes,2);

% Figure out neighbors for both communication types
Connectivity_Matrix(:,:,RADIO_INDEX) = squareform(pdist(Node_Positions(:,1)));
Connectivity_Matrix(:,:,ACOUSTIC_INDEX) = squareform(pdist(Node_Positions(:,1:2)));
for i = 1:length(Connectivity_Matrix(:,:,RADIO_INDEX))
    for j = 1:length(Connectivity_Matrix(:,:,RADIO_INDEX))
        if (Connectivity_Matrix(i,j,RADIO_INDEX) > RADIO_RANGE)
            Connectivity_Matrix(i,j,RADIO_INDEX) = 0;
        end
        if (Connectivity_Matrix(i,j,ACOUSTIC_INDEX) > ACOUSTIC_RANGE)
            Connectivity_Matrix(i,j,ACOUSTIC_INDEX) = 0;
        end
    end
end

% Ensure logical form to matrix instead of numeric
Connectivity_Matrix = logical(Connectivity_Matrix);

clear i j



% Hard wiring for thesis right now
%
%

% NOTE: Read pointer and write pointer are forbidden from being the same
%       except at the start where they are both 1. After that, they are never
%       allowed to have the same value. This simplifies modification logic
%       but requires us to have a queue length 1 more than necessary to make
%       up for the "loss" of a cell when they could be equal.


% Queue length suggested to be number of nodes or greater
if ~(exist('queueLength','var'))
  queueLength = numberNodes;
end

% Following Nick's lead, packets will be represented as
% a 1x4 cell array, with each field being: Dst, Src, Protocl, Data
%      Packets:
%                    Dst      Src      Protocol       Data
% Packet{1x4} =      {}       {}       {}             {}
Packet = cell(1,4);
```

```
% the following matrices are "two pages" each. Page 1 is radio, page 2 is
% acoustic.

txQueue = cell(numberNodes,queueLength + 1,2);
rxQueue = cell(numberNodes,queueLength + 1,2);
% Implementation of buffering.
% eavery node will read from its rxQueue, and write to its txQueue
% another script will handle the juggling of messages from tx to rx.

txQueuePointers = ones(numberNodes,2,2);
rxQueuePointers = ones(numberNodes,2,2);
% these arrays will store read and write pointers
% as usual, the row indicates the node being referenced
% I suggest the convention of [read, write], txQueuePointers[1,1] is
%  txQueue read pointer for node 1.


% there is nothing wrong with leaving queues empty for now. Cells
% are basically pre-allocated blocks of memory. Filling them now or later
% makes no difference. A variable may only take a few bytes, but the overhead
% required of a cell could make storing that value in a cell a matter of
% kilobytes.

messagesSentAndReceived = zeros(numberNodes,2,2);
```

A.2 Main Body

```
Start_Positions = Node_Positions;

energyUsed = zeros(numberNodes,7);
timeElapsed = cell(numberNodes,1);

SimulationOver = false;
success = false;

firstPass = true;

if mode == 5
  path = centralized_shortest_path_algorithm(...
    1,numberNodes,Connectivity_Matrix(:,:,RADIO_INDEX),Node_Positions,0);
elseif mode == 6
  path = centralized_shortest_path_algorithm(...
    1,numberNodes,Connectivity_Matrix(:,:,RADIO_INDEX),Node_Positions,1);
else
  path = modalGetNextNeighbor(...
    1,numberNodes,Connectivity_Matrix,Node_Positions,mode);
end
```

```
step = 0;
timeout = tic;
while ~SimulationOver
    if toc(timeout) > numberNodes % allows for ~8 minutes on 100 nodes
       break                      % which is massive overkill.
     end
  step = step + 1;
  sendAndReceiveMessages;
  for i=1:numberNodes
    energyUsed(i,2) = energyUsed(i,2) + PROCESSING_ENERGY;
    singleNodeTime = tic;
    % Read Messages, node by node and
    % Act on messages

    % Node 1 needs to send out rise commands if path ~= []
    if i == 1 && ~isempty(path)
      path_length = length(path);
      queue_length = length(txQueue(1,:,2));
      if path_length < queue_length
        for j =1:path_length
          rise_packet = Packet;
          rise_packet{1} = path(j);
          rise_packet{2} = i;
          rise_packet{3} = 2;
          [txQueue(i,:,ACOUSTIC_INDEX),txQueuePointers(i,1,ACOUSTIC_INDEX),txQueuePointers(i,2,ACOUSTIC_INDEX)] =...
            add2buffer(txQueue(i,:,ACOUSTIC_INDEX),...
            txQueuePointers(i,1,ACOUSTIC_INDEX),...
            txQueuePointers(i,2,ACOUSTIC_INDEX),...
            rise_packet);
          energyUsed(i,6) = energyUsed(i,6) + 24*ACOUSTIC_SEND_ENERGY;
          % no data, just 3 bytes (3*8) sent
          % this ignores any overheard
        end
        path = [];
      else
        for j=1:queue_length
          rise_packet = Packet;
          rise_packet{1} = path(j);
          rise_packet{2} = i;
          rise_packet{3} = 2;
          [txQueue(i,:,ACOUSTIC_INDEX), txQueuePointers(i,1,ACOUSTIC_INDEX), txQueuePointers(i,2,ACOUSTIC_INDEX)] =...
            add2buffer(txQueue(i,:,ACOUSTIC_INDEX),...
            txQueuePointers(i,1,ACOUSTIC_INDEX),...
            txQueuePointers(i,2,ACOUSTIC_INDEX),...
            rise_packet);
          energyUsed(i,6) = energyUsed(i,6) + 24*ACOUSTIC_SEND_ENERGY;
        end
        path(1:j) = [];
```

```
  end
end
for j=[RADIO_INDEX ACOUSTIC_INDEX]
  [pkt, rxQueuePointers(i,1,j), rxQueue(i,:,j)] =...
    readFromBuffer(rxQueue(i,:,j), rxQueuePointers(i,1,j), rxQueuePointers(i,2,j));
  while ~isempty(pkt)
    % handle packet
    switch j
      case RADIO_INDEX
        energyUsed(i,j+2) =...
          energyUsed(i,j+2) + RADIO_RECEIVE_ENERGY*(3*8 + 8*length(pkt{4}));
      case ACOUSTIC_INDEX
        energyUsed(i,j+2) =...
          energyUsed(i,j+2) + ACOUSTIC_RECEIVE_ENERGY*(3*8 + 8*length(pkt{4}));
    end
    switch determineNextAction(pkt,i)
      case 0 % Do nothing
      case 1 % rise
        % determine change needed, and log delta-Z
        if ~(Node_Positions(i,3) == 0)
          if mode ~= 8
            energyUsed(i,7) =...
              energyUsed(i,7) + abs(Node_Positions(i,3)*WINCH_ENERGY);
            Node_Positions(i,3) = 0;
          else
            if i == numberNodes
              next = i;
            else
              next = greedy_look_back_algorithm(...
                i,numberNodes,pkt{4}(end),Node_Positions,Connectivity_Matrix(:,:,RADIO_INDEX));
            end
            if i ~= next
              rise_packet = Packet;
              rise_packet{1} = next;
              rise_packet{2} = i;
              %                 rise_packet{2} = pkt{4}(end); % forge the message, basically
              rise_packet{3} = 5;
              [txQueue(i,:,ACOUSTIC_INDEX),txQueuePointers(i,1,ACOUSTIC_INDEX),txQueuePointers(i,2,ACOUSTIC_INDEX)] =...
                add2buffer(txQueue(i,:,ACOUSTIC_INDEX),...
                txQueuePointers(i,1,ACOUSTIC_INDEX),...
                txQueuePointers(i,2,ACOUSTIC_INDEX),...
                rise_packet);

              energyUsed(i,6) = energyUsed(i,6) + 24*ACOUSTIC_SEND_ENERGY;
            else
              energyUsed(i,7) =...
                energyUsed(i,7) + abs(Node_Positions(i,3)*WINCH_ENERGY);
              Node_Positions(i,3) = 0;
```

```
      end
    end
  end
  % modify power consumed
case 2 % forward
  % this case is a bare "pass it on." Rebroadcast.
  pkt{4} = [pkt{2} pkt{4}];
  pkt{2} = i;
  % we've now set the fields for the packet, and need to add it to
  % the appropriate txQueue.
  [txQueue(i,:,j),txQueuePointers(i,1,j),txQueuePointers(i,2,j)] =...
    add2buffer(txQueue(i,:,j),...
    txQueuePointers(i,1,j),...
    txQueuePointers(i,2,j),...
    pkt);
  % the packet is officially "handled"
  switch j
    case RADIO_INDEX
      energyUsed(i,j+2) =...
        energyUsed(i,j+2) + RADIO_SEND_ENERGY*(3*8 + 8*length(pkt{4}));
    case ACOUSTIC_INDEX
      energyUsed(i,j+2) =...
        energyUsed(i,j+2) + ACOUSTIC_SEND_ENERGY*(3*8 + 8*length(pkt{4}));
  end
case 3 % payload-forward, send rise command and store for sending
  if ~(mode == 5 || mode == 6)
    rise_pkt = Packet;
    rise_pkt{1} =...
      modalGetNextNeighbor(i,pkt{1},Connectivity_Matrix,Node_Positions,mode);
    rise_pkt{2} = i;
    rise_pkt{3} = 2; % rise code
    % we've now set the fields for the packet, and need to add it to
    % the appropriate txQueue.
    [txQueue(i,:,ACOUSTIC_INDEX),txQueuePointers(i,1,ACOUSTIC_INDEX),txQueuePointers(i,2,ACOUSTIC_INDEX)] =...
      add2buffer(txQueue(i,:,ACOUSTIC_INDEX),...
      txQueuePointers(i,1,ACOUSTIC_INDEX),...
      txQueuePointers(i,2,ACOUSTIC_INDEX),...
      rise_pkt);
    energyUsed(i,6) = energyUsed(i,6) + 24*ACOUSTIC_SEND_ENERGY;
  end
  if i == 48
    true;
  end
  pkt{2} = i;
  payloadDelayedSend{i}{end} = pkt;

  % need to put pkt in to payload delayed-sending
case 4 % end sim
```

```
            SimulationOver = true;
            break
          case 5
            % mode must be 8, and we were forced to rise
            energyUsed(i,7) =...
              energyUsed(i,7) + abs(Node_Positions(i,3)*WINCH_ENERGY);
            Node_Positions(i,3) = 0;
          otherwise
            error('Invalid action chosen. Not implemented yet?');
        end
        % grab next packet
        [pkt, rxQueuePointers(i,1,j), rxQueue(i,:,j)] =...
          readFromBuffer(rxQueue(i,:,j), rxQueuePointers(i,1,j), rxQueuePointers(i,2,j));
      end
      if SimulationOver
        break
      end
    end
    if SimulationOver
      break
    end
    if ~isempty(payloadDelayedSend{i}{1})
      [txQueue(i,:,RADIO_INDEX),txQueuePointers(i,1,RADIO_INDEX),txQueuePointers(i,2,RADIO_INDEX)] =...
        add2buffer(txQueue(i,:,RADIO_INDEX),...
        txQueuePointers(i,1,RADIO_INDEX),...
        txQueuePointers(i,2,RADIO_INDEX),...
        payloadDelayedSend{i}{1});
      energyUsed(i,5) =...
        energyUsed(i,5) + (24+8*length(payloadDelayedSend{i}{1}{4}))*RADIO_SEND_ENERGY;
    end
    timeElapsed{i} = [timeElapsed{i} toc(singleNodeTime)];
  end
  if firstPass
    firstPass = false;
    pkt = Packet;
    pkt{1} = numberNodes;
    pkt{2} = 1;
    pkt{3} = 0;
    payloadDelayedSend{1}{end} = pkt;
    energyUsed(1,7) =...
      energyUsed(1,7) + abs(Node_Positions(1,3)*WINCH_ENERGY);
    Node_Positions(1,3) = 0;
    % log power & delta-z
  end
end
totalTimeElapsed = toc(timeout);

for i=1:numberNodes
```

```
  energyUsed(i,1) = sum(energyUsed(i,:));
end

if SimulationOver
  success = true;
end
```

A.3 Helper Functions

```
function [out_queue, r_ptr, w_ptr] = add2buffer(in_queue, read_ptr, write_ptr, pkt)
% Inputs:
%  queue: the queue to add a message to (row of either tx or rx queue)
%  r/w pointers: the read and write pointers for that queue-row
%  pkt: the 1x4 cell matrix representing a packet to be added
%
% Outputs
%  queue: the modified queue row
%  write_ptr: the next spot to write to
%  read_ptr: the next spot to read from
%

% NOTE: Read pointer and write pointer are forbidden from being the same
%       except at the start where they are both 1. After that, they are never
%       allowed to have the same value. This simplifies modification logic
%       but requires us to have a queue length 1 more than necessary to make
%       up for the "loss" of a cell when they could be equal.

queueLength = length(in_queue);

out_queue = in_queue;

w_ptr = write_ptr;
r_ptr = read_ptr;

if w_ptr == queueLength
  out_queue{w_ptr} = pkt;
  w_ptr = 1;
  if r_ptr == 1
    r_ptr = r_ptr + 1;
   % reset w_ptr to the beginning of the queue, and advance r_ptr if needed
  end
elseif w_ptr == r_ptr
 % we are at the start of simulation
  out_queue{w_ptr} = pkt;
  w_ptr = w_ptr + 1;
```

```
else
  % w_ptr isn't at the end of the queue
  out_queue{w_ptr} = pkt;
  w_ptr = w_ptr + 1;
  if w_ptr == r_ptr
    r_ptr = r_ptr + 1;
    if r_ptr > queueLength
      r_ptr = 1;
    end
  end
end

if isempty(out_queue{r_ptr})
  if w_ptr == 1
    r_ptr = length(out_queue);
  else
    r_ptr = w_ptr - 1;
  end
end



%%add2buffer function
%%adds a packet to the queue given
%%write_ptr indicates what spot in the queue to write to
%
%
%
%%if the write pointer overlaps the read pointer, then a warning is thrown
% if write_ptr == read_ptr
%     warning('The write_ptr is the same value as the read_ptr, this write may overwrite valid data');
% end
%
%%writing the packet to the queue
% queue{write_ptr} = pkt;
%
%%incrementing the write_ptr
% write_ptr = write_ptr + 1;
%
%%if the pointer goes over the max size of the buffer it loops back to the
%%first index
% if write_ptr > size(queue)
%     write_ptr = 1;
% end

end



function path = centralized_shortest_path_algorithm(self_ID, dst_ID,...
```

```
  Connectivity, Node_Pos, Weighted_or_not)

NodeCount = size(Connectivity);
NodeCount = NodeCount(1);

if Weighted_or_not
  Weights = -Connectivity;
  for i=1:NodeCount
    for j=1:NodeCount
      Weights(i,j) = Weights(i,j)*Node_Pos(j,3);
    end
  end
else
  Weights = +Connectivity;
end

for i=1:NodeCount
  Weights(i,i) = inf;
end

[~,path] = dijkstra(Connectivity,Weights,self_ID,dst_ID);
end



function action = determineNextAction(pkt, id)
% only send a packet if it requires "cross talk"
if (id < pkt{1} && id < pkt{2}) || (id > pkt{1} && id > pkt{2})
  action = 0;
  return
end



% if the packet is for us: rise or data
if pkt{1} == id
  if pkt{3} == 2 % if rise packet for us
    action = 1;
  elseif pkt{3} == 0 % data packet for us
    action = 4;
  elseif pkt{3} == 5
    action = 5;
  end
  return
end

% pass packet to forwarding logic
if pkt{3} == 0 % data packet, needs payload-forwarding
  action = 3; % force-rise, don't check for better route
else
```

```
    action = 2; % standard forwarding for everything else.
end

end


function [costs,paths] = dijkstra(AorV,xyCorE,SID,FID,iswaitbar)
%DIJKSTRA Calculate Minimum Costs and Paths using Dijkstra's Algorithm
%
%   Inputs:
%     [AorV] Either A or V where
%         A    is a NxN adjacency matrix, where A(I,J) is nonzero (=1)
%                if and only if an edge connects point I to point J
%                NOTE: Works for both symmetric and asymmetric A
%         V    is a Nx2 (or Nx3) matrix of x,y,(z) coordinates
%     [xyCorE] Either xy or C or E (or E3) where
%         xy   is a Nx2 (or Nx3) matrix of x,y,(z) coordinates (equivalent to V)
%                NOTE: only valid with A as the first input
%         C    is a NxN cost (perhaps distance) matrix, where C(I,J) contains
%                the value of the cost to move from point I to point J
%                NOTE: only valid with A as the first input
%         E    is a Px2 matrix containing a list of edge connections
%                NOTE: only valid with V as the first input
%         E3   is a Px3 matrix containing a list of edge connections in the
%                first two columns and edge weights in the third column
%                NOTE: only valid with V as the first input
%     [SID] (optional) 1xL vector of starting points
%         if unspecified, the algorithm will calculate the minimal path from
%         all N points to the finish point(s) (automatically sets SID = 1:N)
%     [FID] (optional) 1xM vector of finish points
%         if unspecified, the algorithm will calculate the minimal path from
%         the starting point(s) to all N points (automatically sets FID = 1:N)
%     [iswaitbar] (optional) a scalar logical that initializes a waitbar if nonzero
%
%   Outputs:
%     [costs] is an LxM matrix of minimum cost values for the minimal paths
%     [paths] is an LxM cell array containing the shortest path arrays
%
%   Revision Notes:
%     (4/29/09) Previously, this code ignored edges that have a cost of zero,
%     potentially producing an incorrect result when such a condition exists.
%     I have solved this issue by using NaNs in the table rather than a
%     sparse matrix of zeros. However, storing all of the NaNs requires more
%     memory than a sparse matrix. This may be an issue for massive data
%     sets, but only if there are one or more 0-cost edges, because a sparse
%     matrix is still used if all of the costs are positive.
%
%   Note:
```

```
%     If the inputs are [A,xy] or [V,E], the cost is assumed to be (and is
%       calculated as) the point-to-point Euclidean distance
%     If the inputs are [A,C] or [V,E3], the cost is obtained from either
%       the C matrix or from the edge weights in the 3rd column of E3
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using [A,xy] inputs
%       n = 7; A = zeros(n); xy = 10*rand(n,2)
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       IJ = I + n*(J-1); A(IJ) = 1
%       [costs,paths] = dijkstra(A,xy)
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using [A,C] inputs
%       n = 7; A = zeros(n); xy = 10*rand(n,2)
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       IJ = I + n*(J-1); A(IJ) = 1
%       a = (1:n); b = a(ones(n,1),:);
%       C = round(reshape(sqrt(sum((xy(b,:) - xy(b',:)).^2,2)),n,n))
%       [costs,paths] = dijkstra(A,C)
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using [V,E] inputs
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]); E = [I(:) J(:)]
%       [costs,paths] = dijkstra(V,E)
%
%   Example:
%       % Calculate the (all pairs) shortest distances and paths using [V,E3] inputs
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]);
%       D = sqrt(sum((V(I(:),:) - V(J(:),:)).^2,2));
%       E3 = [I(:) J(:) D]
%       [costs,paths] = dijkstra(V,E3)
%
%   Example:
%       % Calculate the shortest distances and paths from the 3rd point to all the rest
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]); E = [I(:) J(:)]
%       [costs,paths] = dijkstra(V,E,3)
%
%   Example:
%       % Calculate the shortest distances and paths from all points to the 2nd
```

```
%       n = 7; A = zeros(n); xy = 10*rand(n,2)
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       IJ = I + n*(J-1); A(IJ) = 1
%       [costs,paths] = dijkstra(A,xy,1:n,2)
%
%   Example:
%       % Calculate the shortest distance and path from points [1 3 4] to [2 3 5 7]
%       n = 7; V = 10*rand(n,2)
%       I = delaunay(V(:,1),V(:,2));
%       J = I(:,[2 3 1]); E = [I(:) J(:)]
%       [costs,paths] = dijkstra(V,E,[1 3 4],[2 3 5 7])
%
%   Example:
%       % Calculate the shortest distance and path between two points
%       n = 1000; A = zeros(n); xy = 10*rand(n,2);
%       tri = delaunay(xy(:,1),xy(:,2));
%       I = tri(:); J = tri(:,[2 3 1]); J = J(:);
%       D = sqrt(sum((xy(I,:)-xy(J,:)).^2,2));
%       I(D>0.75,:) = []; J(D>0.75,:) = [];
%       IJ = I + n*(J-1); A(IJ) = 1;
%       [cost,path] = dijkstra(A,xy,1,n)
%       gplot(A,xy,'k.:'); hold on;
%       plot(xy(path,1),xy(path,2),'ro-','LineWidth',2); hold off
%       title(sprintf('Distance from 1 to 1000 = %1.3f',cost))
%
% Web Resources:
%   <a href="http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm">Dijkstra's Algorithm</a>
%   <a href="http://en.wikipedia.org/wiki/Graph_%28mathematics%29">Graphs</a>
%   <a href="http://en.wikipedia.org/wiki/Adjacency_matrix">Adjacency Matrix</a>
%
% See also: gplot, gplotd, gplotdc, distmat, ve2axy, axy2ve
%
% Author: Joseph Kirk
% Email: jdkirk630@gmail.com
% Release: 1.1
% Date: 4/29/09

% Process Inputs
error(nargchk(2,5,nargin));
all_positive = 1;
[n,nc] = size(AorV);
[m,mc] = size(xyCorE);
[E,cost] = processInputs(AorV,xyCorE);
if nargin < 5
    iswaitbar = 0;
end
if nargin < 4
```

```
    FID = (1:n);
end
if nargin < 3
    SID = (1:n);
end
if max(SID) > n || min(SID) < 1
    eval(['help ' mfilename]);
    error('Invalid [SID] input. See help notes above.');
end
if max(FID) > n || min(FID) < 1
    eval(['help ' mfilename]);
    error('Invalid [FID] input. See help notes above.');
end

isreversed = 0;
if length(FID) < length(SID)
    E = E(:,[2 1]);
    cost = cost';
    tmp = SID;
    SID = FID;
    FID = tmp;
    isreversed = 1;
end

L = length(SID);
M = length(FID);
costs = zeros(L,M);
paths = num2cell(nan(L,M));

% Find the Minimum Costs and Paths using Dijkstra's Algorithm
if iswaitbar, wbh = waitbar(0,'Please Wait ...'); end
for k = 1:L
    % Initializations
    if all_positive, TBL = sparse(1,n); else TBL = NaN(1,n); end
    min_cost = Inf(1,n);
    settled = zeros(1,n);
    path = num2cell(nan(1,n));
    I = SID(k);
    min_cost(I) = 0;
    TBL(I) = 0;
    settled(I) = 1;
    path(I) = {I};

    while any(~settled(FID))
        % Update the Table
        TAB = TBL;
        if all_positive, TBL(I) = 0; else TBL(1) = NaN; end
        nids = find(E(:,1) == I);
```

```
        % Calculate the Costs to the Neighbor Points and Record Paths
        for kk = 1:length(nids)
            J = E(nids(kk),2);
            if ~settled(J)
                c = cost(I,J);
                if all_positive, empty = ~TAB(J); else empty = isnan(TAB(J)); end
                if empty || (TAB(J) > (TAB(I) + c))
                    TBL(J) = TAB(I) + c;
                    if isreversed
                        path{J} = [J path{I}];
                    else
                        path{J} = [path{I} J];
                    end
                else
                    TBL(J) = TAB(J);
                end
            end
        end

        if all_positive, K = find(TBL); else K = find(~isnan(TBL)); end
        % Find the Minimum Value in the Table
        N = find(TBL(K) == min(TBL(K)));
        if isempty(N)
            break
        else
            % Settle the Minimum Value
            I = K(N(1));
            min_cost(I) = TBL(I);
            settled(I) = 1;
        end
    end
    % Store Costs and Paths
    costs(k,:) = min_cost(FID);
    paths(k,:) = path(FID);
    if iswaitbar, waitbar(k/L,wbh); end
end
if iswaitbar, close(wbh); end

if isreversed
    costs = costs';
    paths = paths';
end

if L == 1 && M == 1
    paths = paths{1};
end

% ------------------------------------------------------------------
```

```
function [E,C] = processInputs(AorV,xyCorE)
    C = sparse(n,n);
    if n == nc
        if m == n
            if m == mc % Inputs: A,cost
                A = AorV;
                A = A - diag(diag(A));
                C = xyCorE;
                all_positive = all(C(logical(A)) > 0);
                E = a2e(A);
            else % Inputs: A,xy
                A = AorV;
                A = A - diag(diag(A));
                xy = xyCorE;
                E = a2e(A);
                D = ve2d(xy,E);
                all_positive = all(D > 0);
                for row = 1:length(D)
                    C(E(row,1),E(row,2)) = D(row);
                end
            end
        else
            eval(['help ' mfilename]);
            error('Invalid [A,xy] or [A,cost] inputs. See help notes above.');
        end
    else
        if mc == 2 % Inputs: V,E
            V = AorV;
            E = xyCorE;
            D = ve2d(V,E);
            all_positive = all(D > 0);
            for row = 1:m
                C(E(row,1),E(row,2)) = D(row);
            end
        elseif mc == 3 % Inputs: V,E3
            E3 = xyCorE;
            all_positive = all(E3 > 0);
            E = E3(:,1:2);
            for row = 1:m
                C(E3(row,1),E3(row,2)) = E3(row,3);
            end
        else
            eval(['help ' mfilename]);
            error('Invalid [V,E] inputs. See help notes above.');
        end
    end
end
```

```
    % Convert Adjacency Matrix to Edge List
    function E = a2e(A)
        [I,J] = find(A);
        E = [I J];
    end

    % Compute Euclidean Distance for Edges
    function D = ve2d(V,E)
        VI = V(E(:,1),:);
        VJ = V(E(:,2),:);
        D = sqrt(sum((VI - VJ).^2,2));
    end
end



function id = greedy_farthest_algorithm(self_ID,conn,dst_id,Node_Pos)
% pass in the destination ID, and the row of the present node
% in the connectivity matrix.
id = 0;
Conn_quick_ref = [];
for i =1:length(conn)
  if i == self_ID;
    continue
  end
  if conn(i)
    % if the nodes are connected,
    % store that index, we'll use it later
    Conn_quick_ref = [Conn_quick_ref i];
  end
end
% we have a list of all nodes that are connected
% and we can reference their place in the position matrix
Pos = zeros(length(Conn_quick_ref)+1,1);
for i = 1:(length(Conn_quick_ref))
  % starting at index 2, going to the end of the quick ref
  Pos(i+1,:) = Node_Pos(Conn_quick_ref(i),1);
end
Pos(1,:) = Node_Pos(dst_id,1);
% we've built the array of positions, we need to calculate distance
dist = squareform(pdist(Pos));
% this is a time-saver, but not a space-saver.
% we can look at the first-column alone, dist(2:len,1), basically
% dist(1,1) == 0, by definition, it's the distance between the
% destination and itself.
tmp_distance = inf;
for i = 1:length(dist(:,1))
  if i == 1
    continue
```

```
  end
  if dist(i,1) < tmp_distance
    % if the distance is the least we've found
    % select that index and distance
    % that index can then be used to select from the quick ref.
    tmp_distance = dist(i,1);
    id = i;
  end
end
if id == 0
  error('Could not find any neighbors, which is not likely.\n')
else
  id = Conn_quick_ref(id - 1);
end
% we selected the min distance between neighbor and destination
% and id now contains the id of the appropriate neighbor

end



function id = greedy_shallowest_algorithm(self_ID,conn,dst_id,Node_Pos)
      % pass in the destination ID, and the row of the present node
      % in the connectivity matrix.
      id = 0;
      Conn_quick_ref = [];
      for i =1:length(conn)
        if i == self_ID;
          continue
        end
        if conn(i)
          % if the nodes are connected,
          % store that index, we'll use it later
          if i == dst_id
            id = i;
            return
          else
            Conn_quick_ref = [Conn_quick_ref i];
          end
        end
      end
      % we have a list of all nodes that are connected
      % and we can reference their place in the position matrix
      Pos = zeros(length(Conn_quick_ref)+2,3);
      for i = 1:(length(Conn_quick_ref))
        % starting at index 2, going to the end of the quick ref
        Pos(i+2,:) = Node_Pos(Conn_quick_ref(i),:);
      end
      Pos(1,:) = Node_Pos(dst_id,:);
```

```
      Pos(2,:) = Node_Pos(self_ID,:);
      % we've built the array of positions, we need to calculate distance
      dist = squareform(pdist(Pos(:,1:2)));
      % now we need to select the neighbor that has the shallowest depth
      % depth_iter = Pos(2,3);
      depth_iter = -inf;
      for i=3:length(dist(1,:))
        if (dist(i,1) < dist(2,1)) && (depth_iter <= Pos(i,3))
          depth_iter = Pos(i,3);
          id = i;
        end
      end

      if id == 0
        error('Could not find any neighbors, which is not likely.\n')
      else
        id = Conn_quick_ref(id - 2);
      end
      % we selected the min distance between neighbor and destination
      % and id now contains the id of the appropriate neighbor
    end


function nid = modalGetNextNeighbor(self_ID, dst_ID,...
  Connectivity, Node_Pos, mode)


switch mode
  case 1 % farthest acoustic
    nid = greedy_farthest_algorithm(...
      self_ID, Connectivity(self_ID,:,2), dst_ID, Node_Pos);
  case 2 % farthest radio
    nid = greedy_farthest_algorithm(...
      self_ID, Connectivity(self_ID,:,1), dst_ID, Node_Pos);
  case 3 % shallowest acoustic
    nid = greedy_shallowest_algorithm(...
      self_ID, Connectivity(self_ID,:,2), dst_ID, Node_Pos);
  case 4 % shallowest radio
    nid = greedy_shallowest_algorithm(...
      self_ID, Connectivity(self_ID,:,1), dst_ID, Node_Pos);
%   case 5 % unweighted shortest radio path
%     nid = centralized_shortest_path_algorithm(...
%       self_ID, dst_ID, Connectivity(:,:,1), Node_Pos, 0);
%   case 6 % weighted shortest radio path
%     nid = centralized_shortest_path_algorithm(...
%       self_ID, dst_ID, Connectivity(:,:,1), Node_Pos, 1);
  case 7
    nid = greedy_look_ahead_algorithm(...
```

```
        self_ID, dst_ID, Node_Pos, Connectivity(:,:,1));
    case 8
      nid = greedy_farthest_algorithm(...
        self_ID, Connectivity(self_ID,:,1), dst_ID, Node_Pos);
end
end


function [pkt, r_ptr, queue] = readFromBuffer(queue, read_ptr, write_ptr)
r_ptr = read_ptr;


pkt = queue{r_ptr};


queue{r_ptr} = [];


if r_ptr == length(queue)
  if write_ptr ~= 1
    r_ptr = 1;
  end
elseif r_ptr == write_ptr
  % means a read at the start of simulation.
  % nothing to do here.
else
  if write_ptr ~= (r_ptr + 1)
    r_ptr = r_ptr + 1;
  end
end


end



% need to go through all of txQueue, by row, and add to rxQueue
% based on connectivity matrix (and depth for radio).


% handle radio messages
for i=1:numberNodes
  while true
    [pkt, txQueuePointers(i,1,RADIO_INDEX), txQueue(i,:,RADIO_INDEX)] =...
      readFromBuffer(txQueue(i,:,RADIO_INDEX),...
      txQueuePointers(i,1,RADIO_INDEX),...
      txQueuePointers(i,2,RADIO_INDEX));
    if isempty(pkt)
      break
    end
    messagesSentAndReceived(i,1,RADIO_INDEX) =...
      messagesSentAndReceived(i,1,RADIO_INDEX) + 1;
    % this will run the loop until the txQueue is empty
```

```
    for j=1:numberNodes
      % send to all connected peers, if depth is greater than or equal to 0
      if (Connectivity_Matrix(i,j,RADIO_INDEX)...
          && (Node_Positions(i,3) >= 0) && (Node_Positions(j,3) >= 0)...
          && rand <= RADIO_SUCCESS)
        [rxQueue(j,:,RADIO_INDEX),...
          rxQueuePointers(j,1,RADIO_INDEX),...
          rxQueuePointers(j,2,RADIO_INDEX)] = add2buffer(...
          rxQueue(j,:,RADIO_INDEX), rxQueuePointers(j,1,RADIO_INDEX),...
          rxQueuePointers(j,2,RADIO_INDEX), pkt);
        messagesSentAndReceived(j,2,RADIO_INDEX) =...
          messagesSentAndReceived(j,2,RADIO_INDEX) + 1;
      end
    end
  end
end

% handle acoustic messages
for i=1:numberNodes
  while true
    [pkt, txQueuePointers(i,1,ACOUSTIC_INDEX),txQueue(i,:,ACOUSTIC_INDEX)] =...
      readFromBuffer(txQueue(i,:,ACOUSTIC_INDEX),...
      txQueuePointers(i,1,ACOUSTIC_INDEX),txQueuePointers(i,2,ACOUSTIC_INDEX));
    if isempty(pkt)
      break
    end
    messagesSentAndReceived(i,1,ACOUSTIC_INDEX) =...
      messagesSentAndReceived(i,1,ACOUSTIC_INDEX) + 1;
    % this will run the loop until the txQueue is empty
    for j=1:numberNodes
      % send to all connected peers
      if Connectivity_Matrix(i,j,ACOUSTIC_INDEX) && rand <= ACOUSTIC_SUCCESS
        [rxQueue(j,:,ACOUSTIC_INDEX),...
          rxQueuePointers(j,1,ACOUSTIC_INDEX),...
          rxQueuePointers(j,2,ACOUSTIC_INDEX)] = add2buffer(...
          rxQueue(j,:,ACOUSTIC_INDEX), rxQueuePointers(j,1,ACOUSTIC_INDEX),...
          rxQueuePointers(j,2,ACOUSTIC_INDEX), pkt);
        messagesSentAndReceived(j,2,ACOUSTIC_INDEX) =...
          messagesSentAndReceived(j,2,ACOUSTIC_INDEX) + 1;
      end
    end
  end
end
% Now all "txQueue"s are empty and have been pushed in to the "rxQueue"s


% we advance all delayed sending queues now.
delayedSendUpdate;
```

```
UW_Sim_Setup;
Sim_Run;
% [end_positions, sentAndReceived, energyUsed, timeElapsed] =...
%     UW_Sim_Run(Node_Positions, Connectivity_Matrix, txQueue, rxQueue,...
%     txQueuePtr, rxQueuePtr, mode, Packet);
%
% graph_results;
%toc


constantInit;
topologyInit;
connectivityCreate;
delayedSendInit;
CommInit;
```