DATA ROUTING WITH WIRELESS SENSOR NETWORKS

By

Thomas J. Reale III

A Project Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Approved:

____

Christopher Carothers, Project Adviser

Rensselaer Polytechnic Institute
Troy, New York

November 2011
(For Graduation Dec 2011)

CONTENTS

LIST OF FIGURES . . . iii

ACKNOWLEDGMENT . . . iv

ABSTRACT . . . v

1. Introduction and Historical Review . . . 1
   1.1 Wireless Sensor Networks . . . 1
   1.2 Motivation . . . 2
   1.3 Related Work . . . 4
2. Routing Algorithm Analysis . . . 8
   2.1 Cobweb Data Aggregation and Routing Scheme . . . 8
      2.1.1 Benefits . . . 9
      2.1.2 Drawbacks . . . 10
   2.2 Self Selection Routing . . . 10
      2.2.1 Benefits . . . 14
      2.2.2 Drawbacks . . . 14
3. Knowledge Assisted Selection . . . 16
   3.1 Definition . . . 16
   3.2 Benefits . . . 19
   3.3 Drawbacks . . . 20
4. Implementation . . . 21
   4.1 Introduction to ROSS . . . 21
   4.2 Implementation of the Wireless Sensor Network Simulator . . . 22
      4.2.1 Simulator Design . . . 23
5. Results . . . 26
6. Future Improvements and Conclusion . . . 32
   6.1 Future Improvements . . . 32
   6.2 Conclusion . . . 33

LITERATURE CITED 33

APPENDICES

A. Source Code 40
A.1 Description 40
A.2 Top Level Source Files 41
A.3 Cobweb Files 66
A.4 Self Select Files 83
A.5 KAS Files 104

LIST OF FIGURES

2.1 An example of a path traveled using Cobweb routing 10

2.2 An example of a path traveled using Self Selection 12

2.3 An example of a path traveled using Self Selection where data was duplicated 15

3.1 Sink Broadcast Message Contents 16

3.2 Data Received Message Contents 17

3.3 Data Forward Message Contents 18

3.4 An example of a path traveled using KAS routing 19

5.1 Ideal test grid example 26

5.2 Number of Sensors vs Total Transmission Count 27

5.3 Number of Sensors vs Highest Transmission Count for a Sensor 28

5.4 Number of Sensors vs Average Transmit Time 29

5.5 Number of Sensors vs Maximum Transmit Time for a single sensor . . . 29

5.6 Number of Sensors vs Time to get to sink 30

5.7 Percentage of data received at sink 31

5.8 Average Power Remaining at End of Simulation 31

ACKNOWLEDGMENT

Throughout the process of completing my masters degree there have been numerous people who have helped me to accomplish this goal. I would like to thank my advisor, Professor Christopher Carothers for helping to ensure that I was able to do this and providing me the guidance needed to create a useful result. I would like to thank my friends for helping me in the tough moments, but most of all I would like to thank my family, for without them pushing me I definitely would not have been able to complete this thesis.

ABSTRACT

In recent years, wireless sensor networks have been the source of increasing interest for researchers because they gather information from multiple sources at the same time. This technology allows small sensors to be distributed across a geographical region to collect data, which is sent to a main server via a routing algorithm to be compiled and analyzed. In order to obtain the optimum level of performance, researchers must consider three main factors: power usage, adaptability, and physical characteristics.

This thesis examines current trends in wireless sensor networks, introduces a new simulation environment to test out several current routing algorithms, and introduces an extension on a variant of self-selection routing. The simulator is based on the Rensselaer Optimistic Simulation System (ROSS) platform and models the communication patterns between sensors within a network as well as the overall efficiency of the sensor network. Five algorithms: Self Selection V1, Self Selection V2, Cobweb, and SHR-M as well as an extension of self selection incorporating power into the routing algorithm (KAS) are simulated to determine the accuracy of the simulator and analyze trends in various routing algorithms. Results indicated that the simulator was able to verify the results of the algorithms quickly and provide an output format that can be used to analyze the results quickly and efficiently. The algorithm extensionexplicitly takes power into account when routing information to increase network lifetime as compared to the original algorithm.

1. Introduction and Historical Review

1.1 Wireless Sensor Networks

For centuries, human beings have placed great value on the distribution and attainment of knowledge. Whether being used for research, discovering new frontiers and theories, or saving lives, knowledge has been an important part of human life. Before technology made distributing knowledge a simple process, knowledge was mainly passed via word of mouth. As technology improved over time, the speed in which information could be transmitted improved as well. The traditional methods of using word of mouth and the traditional mailing system gave way to telegraph systems and telephones, and later by the Internet and email. On a larger scale, consumer and government researchers have used wireless sensor networks with sensors distributed across wide geographical regions in order to be more thorough and efficient.

The use of wireless sensors can be traced back to the Defense Advanced Research Project Agency's (DARPA) 1980 project on Distributed Sensor Networks a few years after the creation of Advanced Research Projects Agency Network (ARPANET), an early version of the modern internet. At the time of the DARPA project, technology was limited by the size of individual sensors, which were the size of shoe boxes or larger. Eighteen years later, in 1998, sensors became smaller, more sensitive, more powerful, and more efficient. Producing and distributing sensors has became more cost-effective, and modern sensors can be the size of a dust particle. Improved battery technology has also improved the useful life of sensor networks increasing the usability of the system. Transmission of data requires the most power of any operation but is essential in order for data to be sent to an external server. Batteries have increased such that a large number of transmissions can be sent before the batteries die out [1]. Improvements in creating, distributing, and using wireless sensor networks have allowed engineers to design standards such as the 802.15.4 MAC layer protocol [2] and operating systems such as TinyOS [3] for wireless sensor networks.

The number of applications for wireless sensor networks in the modern world can only be limited by a researcher's imagination. Among their many uses are military operations, environmental modeling, monitoring health conditions, traffic control, sensing industrial characteristics, and infrastructure security. As the quest for knowledge and sharing knowledge continues, new uses for wireless sensor networks are created, including the potential to place sensors in the household to ensure the safety of residents.

1.2 Motivation

Wireless sensor networks can be used for a wide variety of research topics including: physical design, power management schemes, security, sensing capability, processing power, and routing schemes. In order to work properly, a sensor needs to be in constant contact with the main server. If this is not possible, the sensor must be able to store the data it obtains until a researcher can retrieve the information at a later time. The most notable issue with wireless sensor networks is the lifetime of the sensor and its subsequent consumption of power. If a sensor needs to be in constant contact with a server, it requires much more power to operate than a sensor that is merely used to obtain and store data. This thesis addresses the flexibility and power management of three routing algorithms in order to determine the most effective method to use a wireless system network.

Modern wireless sensor networks are able to store information in order to maintain data, increase flexibility in communication, and expand the ability to process data. In order to maintain data, sensors in wireless sensor networks can share information among multiple sensors preventing data from being lost if a remote server is unable to communicate with a single sensor. Communication between sensors in the network is more flexible due to various routing algorithms designed for the specific type of data being stored by each sensor. The stored data can then be coded to consolidate the data received by removing duplications and analysis of the data. These advantages are made possible by technological advances and allow researchers to examine numerous topics with greater speed and accuracy.

While data can be stored within each sensor, at some point the data must be

collected at an external location for the data to be used by researchers. In order to transfer data wireless radios are used to transmit information from outlying sensors to a sensor with an external connection (commonly called a sink). This feature allows the communication process to require less power, but also introduces multiple problems which need to be solved algorithmically. The main consideration in sensor networks is power consumption. Hardware and software improvements have been created, such as more efficient batteries and an awake/sleep cycle. Despite these improvements to increase the network lifetime, researchers still design algorithms that take into consideration the amount of power a network requires to operate. If there is not enough power to operate a sensor, the sensor is considered useless for the researcher because it cannot gather, store, or transmit the required data. If a wireless sensor is over-used there is a possibility it will run out of power. As a result, researchers must carefully consider the amount of power the network they are designing requires while designing algorithms. In situations where time is vital the routes transmitting information may be more power efficient, but require extended lengths of time to transfer information to the sink [1].

Along with the algorithmic challenges of designing a wireless system network, there are also physical limitations when transmitting data between sensors. Studies show the stability of a link can vary according to the amount of power remaining in the sensor, as well as differences based on the direction and type of the antenna within the sensor [4–7]. This causes some data to be lost while being sent between sensors. Reliable communication algorithms can overcome this loss but at a cost of extra delay. Another physical limitation of sensor networks is that in order for any algorithm to be scalable to N sensors, a limit of $O(N^{1/4})$ transmissions is required for the algorithm to not severely limit the lifetime of the network [8]. Security issues must also be taken into account in applications where the information obtained is sensitive. In these situations, algorithms designed to distribute encryption keys can be used to provide secure communication of information to the sink [9].

Due to the fact wireless networks are so versatile, engineers and scientists have created various algorithms to address these issues. This thesis intends to introduce a simulator allowing various routing algorithms to be analyzed and to provide an

extension to the SHR-M routing algorithm explicitly incorporating power into the routing algorithm to increase network lifetime. The aggregation and routing schemes of three established routing protocols, Self Selection Routing V1 [10], Self Selection Routing V2, SHR-M, and Cobweb [11] were examined to use as a comparison. The five algorithms were analyzed according to the amount of data transmitted, the average amount of time taken for the data to reach the server, the amount of data redundancy, the estimated lifetime of individual sensors, as well as the network's ability to dynamically route data.

1.3 Related Work

The broad applicability and complexity of wireless sensor networks provided scientists and engineers a wide variety of issues to address. For example, when a sensor network is initialized there are many situations that occur as a result of the random distribution of sensors that negatively effect network performance. Many times when sensors are distributed in an area there are small areas within the network which produce larger than average amounts of data aka "hot spots". Some algorithms attempt to mitigate this issue by dynamically placing the storage sensors as close to these hot spots as possible [12,13]. These algorithms are used where only a few sensors within a network have the capability to store data or it is possible to position a sink within the network. The placement of the storage nodes close to the hot spots will drastically reduce the transmissions by reducing the distance most of the data will travel. This results in an increased network lifetime and can be applied to any viable network.

Although hot spots do occur, it has become a common trend that all sensors in a network have storage capability. In this case hot spots are only an issue if a sensor has filled its storage system and continues to generate data. There are several algorithms and coding schemes that distribute the data between sensors to solve data retrieval, encoding, compression, and loss of nodes in these situations [14, 15]. If these algorithms were applied to a sensor network, transmission count would be reduced and network lifetime could be increased.

After data has been generated and stored in one or more sensors it must be

routed to the sink. One set of algorithms, known as geographical routing, has each sensor using its geographical location to route data towards the location of the sink. If a sensor has knowledge of the location of itself and a sink, it is trivial to figure out the direction data should travel to reach the sink. The majority of the time sensors gain knowledge of their location using global positioning systems (GPS) systems located on the sensors. There are a few issues with geographical routing such as when a hole appears in the network. If there is a region in the middle of the route between two nodes where there are no sensors, the data will reach that section and then have to route around the hole. This increased transmission count caused the data to take longer to reach the sink and increases the transmission count. Additionally, battery life is reduced with the requirement of obtaining a position and possibly updating it if the sensors are mobile [16–18].

Another form of data routing organizes sensors into a tree structure. Tree based networks are initialized when a sink broadcasts its position throughout the network. During this broadcast sensors organize themselves in a tree structure for data transmission. Lower tier sensors transmit to the next higher level and so on until the sink is reached [19–21]. By using a tree structure an ideal path can be used to transmit data to the sink. An issue with the tree structure is that it requires chatty communication to account for any changes made to the network. One way to address this issue is to allow multiple paths to the top tier reducing the requirements of updating routes [22]. This system improves the network lifetime and allows routes to deal with rare network changes but will have trouble with mobile sensors or constant changes within the network.

A step away from tree based algorithms consists of tiered or landmark based algorithms that assign sensors to be on one of two tiers. These sensors can be placed on a tier according to network density or geographically. When a sensor wishes to send information to a sink, it first sends the data to the nearest top tiered sensor. The top tiered sensor then forwards it to other top tiered sensors until the destination section is reached similar to the IS-IS [23] networking routing protocol [24–27]. Benefits include a simple layout and the top tiered sensors could be more powerful allowing for complex data processing and aggregation. On the

other hand, by routing everything through these higher tier sensors they will lose power faster and will cause the data routes to deviate further from the ideal path by larger amounts as each top level sensor fails. This will increase the delay in transmission and reducing the usefulness of the network.

A different set of routing algorithms attempt to contact every sensor in the network to disseminate information [28–30] or to retrieve information [31] to/from every sensor within the network with minimal overhead and duplication. The issue with these algorithms is that is is not easy to reach every sensor without a large amount of overhead or knowledge of geographical locations and transmission range. Algorithms sometimes rely on routes that nears every sensor, but does not require every sensor to transmit information in order to minimize the overhead, but this system cannot completely remove it [29]. These algorithms are very useful for broadcast messages such as when a sink broadcasts its position but are not very useful in the majority of routing instances where point to point communication is desired.

An interesting data dissemination technique consists of rumor routing or ant routing [32, 33]. This kind of routing has a message traversing the network in a pseudo-random fashion. When the message reaches a sensor meeting certain criteria, such as containing information about certain types of events, it transmits data back along the traveled path to a sink. This type of dissemination is very slow and takes a long time for the message to discover every sensor that meets the criteria, but uses very few transmissions and allows for retrieval of only the data that is needed.

Another method of gathering type specific data creates a path to the sink that does not follow an optimal path towards the sink. The message follows the path allowing it to collect the most information possible [34]. The route the data takes is determined by the type of data to be collected. This algorithm may be very useful in gathering data but an unstable sensor network may make it difficult for the message to traverse all sensors that it wishes to. Along the same lines, directed diffusion [35] uses a publish/subscribe algorithm to request specific data from areas in the network. The data is routed back to the sink along the path that the discovery packet took and has similar results.

One important aspect of sensor networks to take into consideration is the

awake/sleep schedules of sensors. Many times sensors are in very low power states and are not available for communication. In these situations routing takes longer at the best case and does not occur in the worst case. One solution that has a lot of promise and can be used in many situations consist of MAC protocols that synchronize the sleep schedules of sensors that are near to each other [36, 37]. Synchronization of sleep cycles with nearby sensors allow for fewer communication delays as a result of unavailable sensors. This improves the performance of algorithms that assume constant contact and use ideal paths when transmitting data. Some of the issue with these systems is communication overhead, but because synchronization allows the best case to occur a greater percentage of the time this communication overhead is negligible.

Different routing algorithms are selected according to the goal of the algorithm. Dissemination routing algorithms are used to broadcast or retrieve data from all sensors within the network, tree based algorithms provide an easy interface to generate an ideal path, ant routing allows a slow gathering of data, and synchronization allows sensors to communicate without delays incurred by sleep cycles. But each of them have their own drawbacks limiting them based on the situation or available power stored on the sensor. Several of the routing algorithms described require point to point communication from one sensor to another and all must be able to handle a constantly changing network due to sleep cycles or mobile sensors. Many of the routing algorithms try to balance time taken, power consumption, or ability to adapt according to the goals of the algorithm. But it is the goal of all algorithms to produce an algorithm that excels at all 3.

2. Routing Algorithm Analysis

In order to display the abilities of the simulator three established routing algorithms are described and analyzed for use in evaluating the simulator as well as to compare against the extension of the SHR-M algorithm.

2.1 Cobweb Data Aggregation and Routing Scheme

The Cobweb Data Aggregation and routing scheme is proposed and described in [11]. The algorithm consists of two phases, a data aggregation phase and transmission phase. The aggregation phase is addressed to reduce duplication when sensors record the same event and reduces the amount of data needed to be sent to the sink. The aggregation scheme is not the focus of this paper it will not be evaluated. Instead, the second phase of transmission is described below and analyzed.

The transmission phase described in the article is as follows:

1. Each sensor must contain some information about it's neighbors in storage. The information consists of three items: Node ID (NodeID), Remaining Energy (RemEnergy), and Hop Count (HopCount) to a sink.

2. When data is generated and aggregated at a single node the transmission process is started. Initially, node A is the node that has the aggregated data.

   (a) If A isn't the neighbor node of the sink node, A finds the neighbor node B satisfying $HopCount_B < HopCount_A$ and $RemEnergy_B$ max among neighbors. A sends the data to B.

   (b) If B isn't the neighbor node of sink node, it does as (b) and sends acknowledgement containing RemEnergy to A.

   (c) B repeats step (a) until sink node is reached.

A few assumptions were made in implementing this algorithm that allow it to perform efficiently and satisfy the description of the algorithm.

Prior to the aggregation and transmission phases, a sensor that wishes to identify as a sink broadcasts its location to all sensors within the network. There are three fields in the broadcast message, `sink_id`, `last_id`, and `hop_count`. The `sink_id` field is the identifier for the sink for maintenance purposes. The `last_id` is the ID of the sensor that last broadcast the location message and is used in conjunction with the `hop_count` at each sensor to learn the best path. The `hop_count` is initially set to 0 at the sink and is incremented by 1 each transmission.

When finding a neighbor with hop count satisfying $HopCount_B < HopCount_A$ the following definition is used:

Definition Given node A with k neighbor nodes $0 \ldots k$ the node B is chosen such that $B = min_0^k HopCount_i$ where i is a number from 0 to k.

The definition indicates that the sensor with the lowest hop count is chosen, not just a hop count less than the current node. This guarantees that the algorithm will use the fewest number of hops to transmit the data to the sink.

The simulation also uses awake/sleep cycles similar to a real sensor network would to reduce power consumption. When sending information to another sensor, if that sensor is asleep, the sender periodically resends data until the neighbor wakes up.

An example of a route that data has traveled in transmitting data from a position in an 11x11 grid, with a sensor transmission radius of 1 is shown in Figure 2.1.

2.1.1 Benefits

The first benefit of this routing scheme is that it guarantees data will be routed to the sink as efficiently and quickly as possible. Both efficiency and speed are obtained by using the fewest number of hops possible. This allows the algorithm to minimize the effect on individual sensors that may not be in the path of the data and reduces the wait time on sensors. Sensors who are not the destination also do not have to pay attention to any transmissions not destined for them.

The energy usage is minimized and considered by using the sensor with the maximum power. This allows alternate paths to be immediately chosen as some

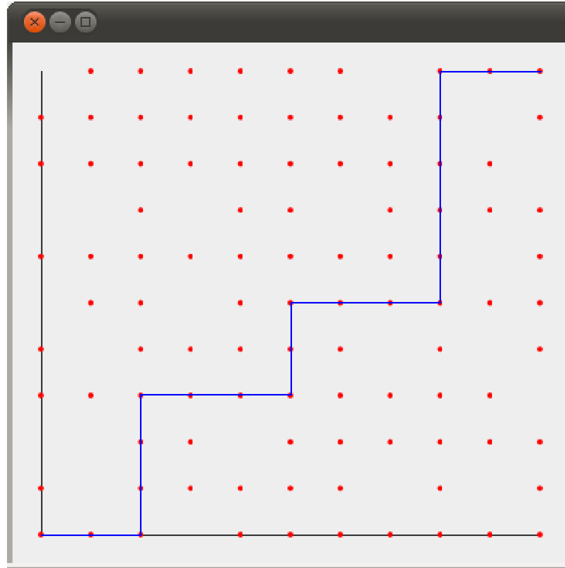

Figure 2.1: An example of a path traveled using Cobweb routing

sensors become overwhelmed.

2.1.2 Drawbacks

Due to the simplistic selection of which neighbor should forward the data there is a significant delay if that neighbor becomes unavailable. This algorithm also does not describe any way to update routes when sensors drop or are asleep. This inability to deal with network changes would not work in mobile or unstable networks.

2.2 Self Selection Routing

The Self Selecting algorithm initially proposed in [10] and enhanced in future versions (SHR/SHR-M [38] and SSR [39]) is a braided [40] multi-path routing algorithm. This algorithm uses a back-off timer to select which sensor forwards the data allowing a path to select itself. The algorithm is intended for a request/response type of system, where the sink sends out a broadcast to all sensors requesting data from an individual sensors, and the sensor replies with information. The broadcast message sent from the sink contains information to let each sensor learn the distance it is from the sink. Once the desired sensor learns about the sink, it is able to

start the transmission process. A summary of the original self selection algorithm is provided here:

1. When a sensor is sending data, it transmits a message containing an expected hop count field along with the data. The expected hop count field is the distance according to the sensor to the sink minus 1.

2. When nearby sensors receive the message, they start a back off timer that is a function of their known hop count and the expected hop count in the message. If the parameters are carefully tuned, the sensor with the shortest distance to the sink will have the smallest back off timer. This sensor will then respond by forwarding the data to it's neighbors.

3. The act of forwarding the data is also used as an ACK message to the originating sensor indicating a sensor has self selected. Because the ACK may not have reached all the sensors who started back off timers, the originating sensor sends an explicit ACK to mitigate duplication.

4. The process repeats until the sink is reached.

An example of routing using the self selection algorithm can be seen in Figure 2.2. This display graphically represents a 11x11 unit geographical region containing sensors. The units of the grid can be coordinated to be consistent with the transmission range of the sensor that is being modeled but the units are generalized for the simulation. Using a transmission range of 1 unit, Figure 2.2 represents the path that data has traveled from the sensor located at position 0,0 to the sensor located at position 10,10. Each blue line indicates that a message has been sent between the two sensors. The Larger green circles are used to indicate the sensors that self selected along the path in this instance.

When data was being forwarded towards the sink and a sensor is calculating the back off timer the original self selection algorithm used Equation 2.1 to obtain the value.

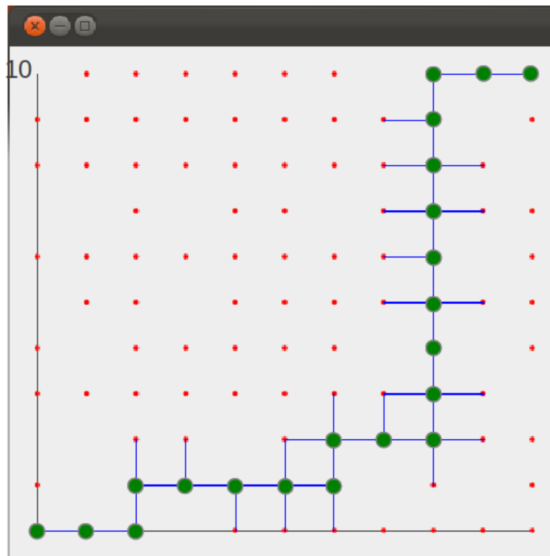

Figure 2.2: An example of a path traveled using Self Selection

$$d_{backoff} = \begin{cases} \lambda * (h_{table} - h_{expected}) * U(0,1) & if\, h_{table} > h_{expected} \\ \frac{\lambda}{h_{expected} - h_{table} + 1} * U(0,1) & if\, h_{table} <= h_{expected} \end{cases} \tag{2.1}$$

In this equation h_{table} represents the stored distance to the destination, $h_{expected}$ is the expected distance received in the message, U is a random number between its arguments, and λ is a tuning parameter to help increase the variability between sensors. One result of this equation is that it is possible for a sensor that is further away from the sink to self select with a probability of $\frac{1}{4}$ due to the random variable in the equation [39]. This feature was meant to allow for self healing routes, or updates to the routing of data when routes are terminated and are no longer the ideal path. This situation occurs when a path that was ideal is cut short by an active (sleep cycle) or passive (node lost power) means. The self healing version (SHR) of the algorithm defines a protocol that updates the hop count within the sensor in order cause future data transmissions to avoid the dead path. When the path had been updated the network had healed and would use the new ideal path to send data without traveling down a dead end.

Another alteration to the original algorithm in the SHR version of the algorithm was a change in the calculation of the back off timer. The Self Healing algorithm used a separate algorithm to perform the healing process and therefore did not need to incorporate it into the back off timer. The back off equation was then altered to be Equation 2.2 thereby removing the possibility that a sensor that is further away will be selected before a sensor that is closer (in the ideal case). This also removes many situations where it is possible for data duplication to occur by increasing the amount of time sensors have to self select.

$$d_{backoff} = \begin{cases} \lambda * ((h_{table} - h_{expected}) * U(0,1) + 1) & if\, h_{table} > h_{expected} \\ \frac{\lambda}{h_{expected} - h_{table} + 1} * U(0,1) & if\, h_{table} <= h_{expected} \end{cases} \quad (2.2)$$

An additional alteration to the original algorithm included synchronization of sensors when any communication passed between sensors. By synchronizing sensors the likelihood that communication with a desired sensor is unsuccessful due to sleep cycles is drastically reduced. This reduces the number of transmissions as a result of sleep cycles and reduces the overall transmission time of information.

One last addition to the algorithm included in the SSR version of the algorithm reduces the delay in transmission to almost zero by using memory about neighboring sensors. When a sensor is transmitting data and a neighbor self selects that neighbor will be remembered for future use. At any point in the future when data is being forwarded to the same destination the sensor will then automatically forward the information directly to the remembered neighbor. This removes all delays due to sensors self selecting except for two situations: the initial self selection and the preferred neighbor being unavailable. The initial self selection process is unavoidable as the sending sensor must learn who is the priority. It is also unlikely that the preferred neighbor will become temporarily unavailable due to the synchronization process. When a sensor does become permanently unreachable the self selection process can occur again and if necessary a healing algorithm can take place reducing any further delays.

2.2.1 Benefits

The benefits of the algorithm lie in it's ability to dynamically adapt to any situation. It does not matter if nodes drop off due to sleep cycles, power loss, or are mobile. It also does not matter if the sink is mobile, as the sink could be broadcasting it's position and the routes could be updated to maintain the ideal path. This is an enormous benefit as sensors are constantly changing states and require awake/sleep cycles in order to last a significant amount of time. The additions of synchronization, self healing, and memory produce an even more robust and dynamic network that only has slight delays when initializing the network and when a network change is required.

2.2.2 Drawbacks

The main detriment to the original algorithm was communication overhead. When forwarding information each sensor that forwards the data must send out two transmissions, the initial send the explicit ACK. This immediately doubles the number of transmissions and will significantly reduce power. Because power is vital in sensor networks this is an issue that is hard to ignore. This issue was addressed in the SSR version by the addition of memory about the ideal path within the sensor. This removed the additional ACK in the majority of cases.

Another drawback in the original algorithm was the amount of data duplication that was produced as a result of the back off equation. Data became duplicated when multiple sensors would self select at the same time. An example of this situation is displayed in Figure 2.3. This situation causes a drastic increase in the number of transmissions and reduces the network lifetime by a factor of the number of duplications. This issue was addressed by the SHR version of the algorithm by altering the equation to reduce the likelihood of data duplication. By increasing the time period that sensors can self select it is less likely that two sensors will select simultaneously. This issue is also addressed in the SSR version of the algorithm after the preferred neighbor is selected. When the preferred neighbor was known data could not be duplicated as it was only directed to the preferred neighbor.

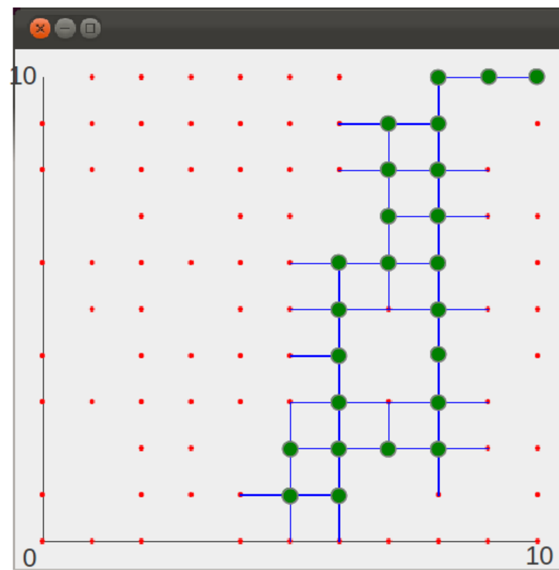

Figure 2.3: An example of a path traveled using Self Selection where data was duplicated

3. Knowledge Assisted Selection

From these algorithms, the Knowledge Assisted Selection (KAS) algorithm has been conceived to provide an extension to the self selection algorithms incorporates many benefits of the self selection algorithms and incorporates power of sensors into the routing algorithm. The KAS selection algorithm consists of a selection algorithm that is assisted by knowledge about a sensors' neighbors. This algorithm guarantees only one path will be used to send data, as well as a dynamic path that can withstand sleep cycles, power changes, and mobile sinks.

The KAS algorithm also does not require complex processing on the knowledge it has about other neighbors. The knowledge learned about neighboring sensors consists only of the number of hops and the power remaining. The processing on these values is very minimal, consisting only of keeping the values sorted.

3.1 Definition

There are two phases to the algorithm, the initial knowledge gathering phase and the transmission/maintenance phase. The knowledge phase initializes the network when a sink identifies itself. This process is accomplished by flooding information about the sink and is used to distribute the information required to forward data among the sensors. The transmission/maintenance phase consists of routing the data to the sink using an algorithm similar to self selection and ensuring the knowledge in each sensor is current.

Knowledge Phase The KAS algorithm initiates when a sink wishes to broadcast its location. The sink broadcasts a simple message that is distributed throughout the network. The contents of this message are shown in Figure 3.1.

sink_gid	num_hops	source_gid	Last_gid	last_power

Figure 3.1: Sink Broadcast Message Contents

A sensor receiving this packet will store or update its knowledge of any sinks based on this information. Because every sensor broadcasts this message, a sensor

can learn the distance of all nearby sensors to the sink (num_hops) and the power that a neighbor currently has (last_power). The sensor sorts and stores the information for use in selecting a preferred neighbor. Neighbors are sorted according to the preferential value P in Equation 3.1.

$$P = hop_count + log_2(\frac{power_max}{power}) \tag{3.1}$$

This increases the value of P by 1 every time the power decreases by half. Once a sensor has received this message, it broadcasts its information to all nearby sensors.

Transmission/Maintenance Phase The transmission phase can be initiated any time after the knowledge phase has reached a sensor that has data stored. This phase is initiated by a data received message. The contents of this message is given in Figure 3.2.

source_gid	nonce	dest_gid	num_nearby	last_power	data

Figure 3.2: Data Received Message Contents

The first two fields in the message uniquely identify the message. The nonce value is a number unique to the source of the data. This allows the sink to categorize the data and to allow sensors in transmit to uniquely identify the data to remove duplicates. The last_power field indicates the power remaining in the sensor that transmitted this message. This allows sensors to stay up to date with nearby sensors without any extra messages and allows the routes to dynamically update based on the current state of the network. The dest_gid field is the sensor that the sender prefers. This is easily achieved by accessing the first element in the sorted array described in the *knowledge phase*. The num_nearby field is the number of sensors that the sender has knowledge of and is used to create an expiration timer in sensors that do not forward the data. The data that this sensor wishes to transmit is attached. When this message is sent the sender starts a short back off timer. This timer is 2*transmission_time long to cover the length of time it would take to respond plus a some extra to allow the neighbor to process other messages needed.

All nearby sensors who are awake and have power will receive this message. At this point there are three scenarios:

1. If the sensor with `dest_gid` receives the message it will immediately forward the data. This accomplishes three things, it moves the data along the path very quickly, acts as an ACK to the sender, and updates the sender with the power remaining.

2. If the timer on the sender expires, it assumes that its preferred sensor is not available and sends a data forward message (Figure 3.3) to the next best sensor. This message is very small and takes almost no time to transmit as the data has already been received at the available sensors. This is repeated in a round robin system until a neighbor forwards the data. If all neighbors have been asked to forward the data and none have responded, the sensor will resend the data to ensure that they neighbors have data to forward.

3. If a sensor that is not the destination receives the message it starts an expiration timer. This timer is calculated as $transmission_time + transmit_delay * num_nearby$. The `transmit_delay` is the time it takes for a small message of a few bytes to be received. During this expiration time, the sensor listens for any communications from the sender. If it does not receive any communications related to the message it deletes the message from it's memory as the data has been forwarded by another sensor.

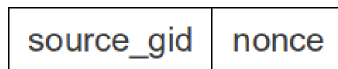

Figure 3.3: Data Forward Message Contents

When the sender receives the data it updates the power from the sensor that forwarded the information and ensures the neighbors are sorted according to priority. This process repeats until the information reaches the sink, at which time the sink sends a data_received message with no data indicating that the sink has been reached.

An example of a route that data has traveled in transmitting data from a position in an 11x11 grid, with a sensor transmission radius of 1 is shown in Figure 3.4.

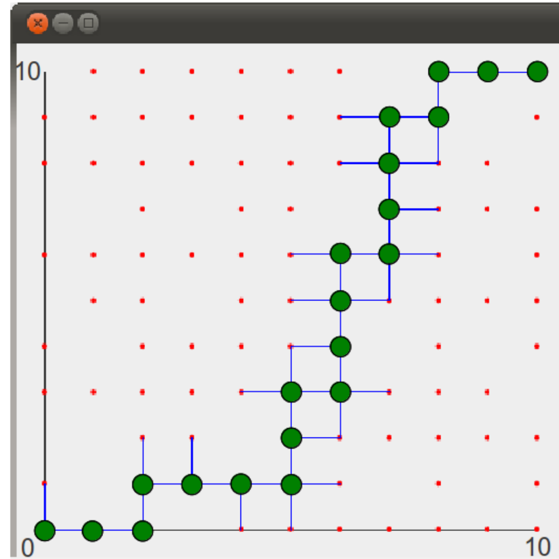

Figure 3.4: An example of a path traveled using KAS routing

3.2 Benefits

The KAS algorithm incorporates many of the advancements made by the SHR, SHR-M, and SSR self selection algorithms and incorporate power usage in an attempt to increase overall network lifetime. KAS routing includes the self selection ability to dynamically adapt to temporary drops in sensors due to sleep cycles and other events. Similar to the original self selection algorithm, the data is only sent once in order to forward the information (unless all neighbors are unavailable). This contrasts the SSR algorithm in situations where the preferred neighbor is not available. KAS uses only one transmission per sensor to forward information similar to the SHR-M version of the algorithm. Each transmission transmits the data to neighbors, acts as an ACK to the previous sensors, and updates the knowledge in neighboring sensors about power remaining. The inclusion of the expiration timer removes the necessity for an explicit ACK to be sent by any sensor and reduces the number of transmissions. One feature the KAS algorithm shares with the SSR

algorithm is fast and consistent transmission times. As long as there are no drops in sensors the data will be sent as quickly as the sensors can transmit and will be received at the destination with minimal delay.

A difference the KAS algorithm has over the self selection algorithms is in the way the neighboring sensor is selected when it is unknown which is the best sensor to forward information. In KAS the initial message that was flooded out by the sink is used to inform all sensors which sensor would be ideal. In all of the self selection algorithms this is done by querying all neighbors and letting them decide which is the ideal neighbor to use.

The inclusion of power allows routes to increase the lifetime of over-used sensors. It is possible for data to be routed around sensors that are in hot spots without any additional overhead. This feature will increase the overall lifetime of the network.

3.3 Drawbacks

An issue with the scheme when compared to the self selection algorithm may be the time it takes to deal with sensor drops. If the equation in the self selection algorithm is properly tuned it may respond faster than the KAS algorithm. The algorithm as it is currently described also has trouble updating paths if sensors are mobile, but as is described in the future improvements section this is easily remedied.

The KAS algorithm also does not currently incorporate a healing mechanism causing data to be trapped if the routes are not updated. This will slightly be mitigated by the loss in power at the sensors as packets continue to be sent to them, but losing sensors is not an ideal situation.

4. Implementation

The simulator used to model the sensor network is designed using the Rensselaer Optimistic Simulation System (ROSS). This simulator is a discrete event simulator designed by Professor Christopher Carothers and his students. The simulation framework is designed using the message passing interface (MPI) allowing simulations to be performed both on home computers as well as IBM Blue Gene supercomputers.

This chapter describes an introduction in developing simulations using ROSS as well as how I used this framework to develop a simulator for sensor networks.

4.1 Introduction to ROSS

As mentioned earlier, ROSS is a discrete event simulator implemented in the C programming language. A discrete event simulation is based off of events occurring at a specific time. For example: if a sensor wishes to send a message to another sensor it creates a message of a predefined structure and tells the ROSS framework to fire that event on the destination sensor at a given time. When the destination sensor reaches that time in it's processing, it receives the event and processes the message.

The implementation is based on a queue of managed events on logical processors (LPs). ROSS then maps these LPs onto physical processors available on the system. Each LP processes it's queue of events in the according to it's notion of time. ROSS can be run in several modes, the most notable of which are sequential and optimistic. Sequential mode forces all sensors to be synchronized with respect to time. This is fairly straightforward and ensures that events occur as expected. On the other hand, optimistic mode allows each LP to process events without regard for other sensors. Optimistic mode allows the LPs to run as quickly as possible, and could make the simulation run much faster but increases coding complexity. Relating it to wireless sensor networks; when a sensor wishes to send a message to another message at a given time, the event needs to be inserted into the LPs queue. If the

destination has already processed events in the future, in order to obtain correct results, it must roll back the changes that were done and then process the received event. This is accomplished by incorporating reverse functions for every event that can occur.

4.2 Implementation of the Wireless Sensor Network Simulator

The basic design characteristics of the Wireless Sensor Network Simulator include:

- Wireless sensors are distributed throughout a two dimensional plane randomly. This is accomplished prior to any simulation events. The area of the plane and number of sensors distributed can be defined at run time. Sensors are distributed such that no two will be placed at the same location.

- The initial power and sensor transmission radius are common to all sensors. The transmission radius is independent of the power.

- There are several events that are time based. The rate at which these events, or the length of time these events occur are subject to a gaussian function with various standard deviations to allow for random differences.

- A sensor designated as a sink is the final destination for all data. In real life situations it may be the case that the sink has a connection to an external server though satellite or wire. For the purpose of this simulation, if that is the case than the sensor must satisfy the conditions that it will never lose power and have a high bandwith to the external server. This removes any transmission time and extra power considerations.

- The physical systems are modeled based off of the MicaZ mote. Documentation for this sensor can be found at [41].

- It is assumed that the wireless sensors use the 802.15.4 protocol to transmit information. This protocol allows for great variability and reliably when

transmitting data from one sensor to another. In order to simulate unexpected delays due to communication collisions, a gaussian distribution is used to alter the length of transmissions.

From these basic constructs, a wireless sensor network simulator has been demonstrated that allows any routing protocol to be implemented quickly and efficiently. It is necessary that some knowledge specific to ROSS be learned from the ROSS documentation and experiments. But a demonstration of how the simulator is designed, and design choices are described in this section.

4.2.1 Simulator Design

There are three basic elements that are consistent throughout the simulator: sensor information, messages, and events. The sensor information must contain information to manage connections, manage paths to the sink, and manage its own status. Messages must contain all possible information for all events that will occur. This is necessary as there is only one message structure can be used in the simulation for optimization purposes. Events consist of two types, messages that are passed between sensors and wait events on the same sensor. There are a couple of ways to implement messages between sensors; two events can be generated according to the beginning and end of the transmission, or only the end of the transmission has an event associated with it. The majority of the events are of the latter variety because a reliable protocol is used and it is not necessary to know the start of transmission only the length of time it took.

The statistics that the simulator measures could be anything that is needed. The statistics evaluated for use in this thesis consisted of transmission time, transmission count for each sensor, time data required to get to sink, time awake, time asleep, power consumption, and number of sensor failures. As the simulation is run the basic operations of the algorithm, such as when messages are sent, update the statistics as each event occurs. At the end of the simulation each sensor is queried for the information specific to that sensor and displayed at the end. These statistics allow a broad variety of trends to be found and demonstrated for comparison with other algorithms.

The basics of the simulator are predetermined by the format that ROSS requires. ROSS requires several functions: init, event, event reverse compile, and final in order to run a simulation. The initialization function is run prior to the simulation on each sensor. This function initializes the variables within a sensor such as setting the initial power and setting the location of a sensor. Once all sensors have been set up an initial event is created. This event establishes the initial relationship between sensors. Due to the fact that the program is a simulation the nearby sensors must be determined by searching through all existing sensors. This is accomplished by the initial event. After this initial event has given all sensors knowledge of their neighbors.

The sleep cycle is also started in the initial event by creating an event in the future indicating a sensor should go to sleep. Each time that the sensor changes states from awake to asleep a separate event is created to change back to the other state and the power and timing statistics are updated. This occurs in the background and can be modified by changing the length of time between events in the current simulation, but further changes can be made to alter how the cycles work.

At this point in the simulation the algorithm initiates. In order to start modeling an algorithm the initial event also creates an event to have a sink broadcast it's location. At this point the algorithm takes over. The aspects of the algorithm are defined as the separate events that are available. This consolidates all information that define an algorithm to one file. All of the message passing is accomplished through functions that are the same among various algorithms.

There are some additional differences between algorithms that require changes to certain algorithms that manage data within a sensor. These methods are incorporated into a utils file allowing management of any algorithm specific information.

As several algorithms have been implemented a very nice feature of the system has been apparent. The requirements of many algorithms are similar including requiring information about previous messages, segments where the sensor waits for responses, and sequences of messages. This allows for all of the utility functions to be similar and allowing the algorithmic differences to be implemented based on given examples.

The design of the initial set up, message passing structure, and examples of the utility functions allow the implementation of a variety of algorithms. This basic framework is the simulation design. This framework allows the majority of the statistics to be recorded with little additional effort.

The sensor information defines the information and can vary from algorithm to algorithm. These changes tend to be similar to other algorithms and can be replicated with little difficulty for other altorithms. The messages passing between sensors along with the associated statistics are handled by the basic framework. Events are the crux of the differences between the algorithms and allow easy differentiation and definition of the algorithms.

The simulator that has been defined is implemented in a way that separates the algorithmic aspects from the basic interface. This allows quick implementation of algorithms and automatic management of many of the statistics.

5. Results

In order to test the simulator to determine its accuracy 5 algorithms were simulated: the cobweb routing algorithm, self selection V1, self selection V2, SHR-M, and KAS. The self selection V1 algorithm is the original self selection algorithm [10], The self selection V2 algorithm is the same as V1 except altering the back off equation to be Equation 2.2, The SHR-M algorithm is defined in [38], and the KAS algorithm is defined in this paper. The ideal system is one where sensors have unlimited power and are always awake. The amount of data generated was enough to take one unit of time to transmit to another sensor. A transmission delay of 0.1 seconds was added to each transmission to allow for any processing time. Every transmission time is approximated using a gaussian distribution with a standard deviation of 0.1 imitating a situation where there is little transmission conflict. Each sensor has a transmission radius of 1 unit. The grid tested was a 11x11 grid with a single sensor generating data located at 0,0 and a single sink located at 10,10. An example grid is displayed in Figure 5.1. The simulation was run for 799 seconds to gain a reasonable average of any statistics that require timing operations.

The first set of images are taken with respect to the number of sensors in the grid. The number of sensors contained within the grid range from 72 to 100.

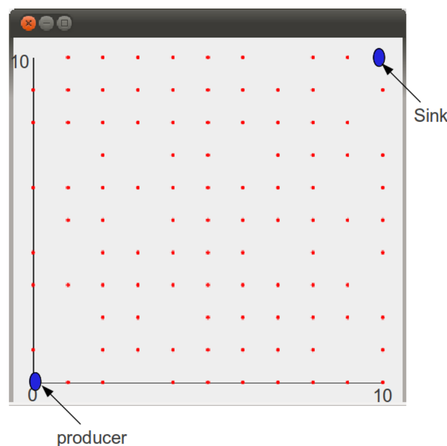

Figure 5.1: Ideal test grid example

As was described earlier, one of the most important aspects of sensor networks

is power consumption. One factor that helps to determine the overall network lifetime is the total transmission count during a simulation. As can be seen in Figure 5.2 the cobweb and KAS algorithms use fewer transmissions than any of the self selection algorithms. This is due to the chat-oriented nature of the self selection algorithm as well as duplications in data. The Cobweb and KAS algorithms only require one transmission in order to forward data, whereas the self selection algorithm requires the data transmission and an ACK message be sent. During the self selection process, it is also possible that multiple sensors will self select causing data duplication. This can easily cause an exponential rise in transmissions if the back-off timer is not tuned properly. The back-off equation for self selection (Equation 2.1) was calculated with a lambda value of 5. This value provided very few duplications in data as viewed during simulations but did provide some.

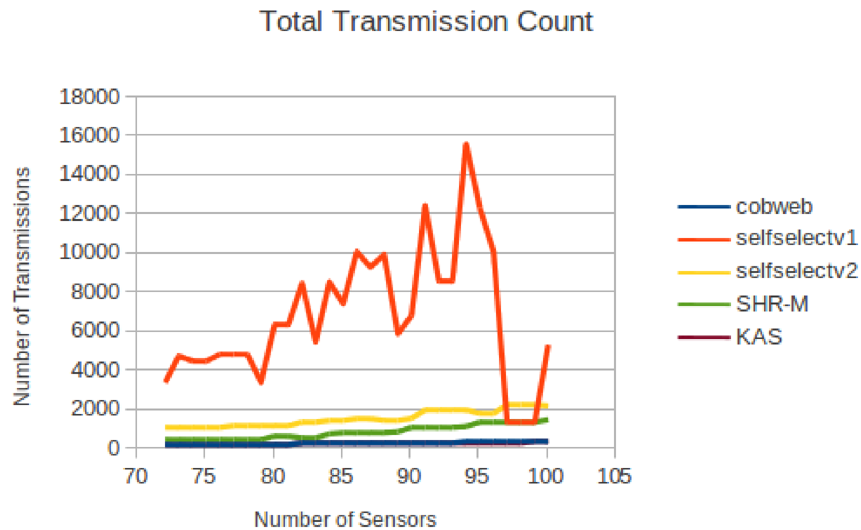

Figure 5.2: Number of Sensors vs Total Transmission Count

The figure also indicates that the cobweb algorithm has slightly fewer transmissions than the KAS algorithm. This is a result of an extra transmission when data reaches the destination. This transmission is required because when the sink receives the data it must send an explicit ACK to conclude the transmission of the data.

Another field that was tracked in coordination with the total number of trans-

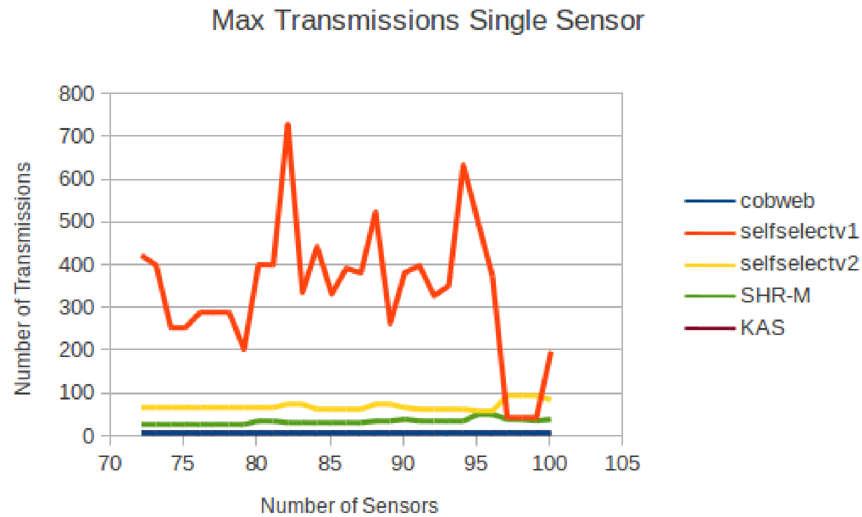

Figure 5.3: Number of Sensors vs Highest Transmission Count for a Sensor

missions was the highest transmission count for a single sensor (Figure 5.3). This statistic gives a sort of standard deviation, allowing a comparison to determine whether sensors along the chosen path will lose power quickly, or whether the path that is taken the majority of the time. The results show that the cobweb algorithm uses slightly more than the assisted algorithm. This is due to the cobweb algorithm always taking the same path, while the assisted may vary slightly based on the power remaining. The self selection algorithm has a much higher number due to the reasons described earlier.

Similar to the transmission count, the transmission time was tracked for the average and maximum values (Figures 5.4 and 5.5). These graphs are consistent with the transmission count.

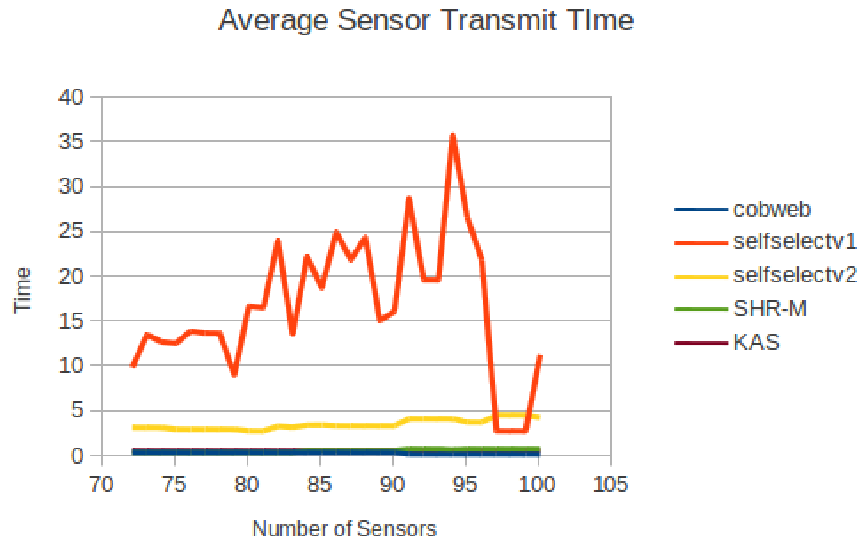

Figure 5.4: Number of Sensors vs Average Transmit Time

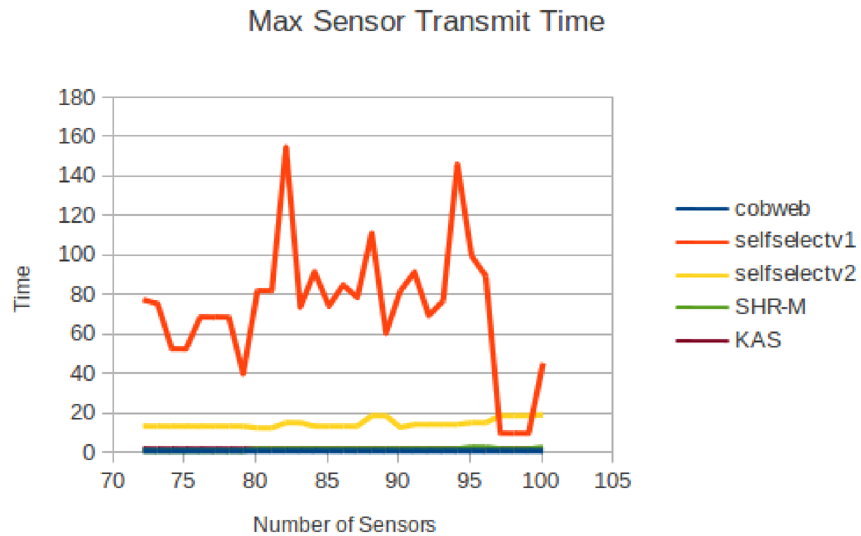

Figure 5.5: Number of Sensors vs Maximum Transmit Time for a single sensor

Another statistic that is very important to end users especially in real time applications is the length of time it takes for data to be transmitted to the sink (Figure 5.6). In this graph the self selection algorithm suffers greatly under the

ideal case as the back-off timer causes a delay at each step.

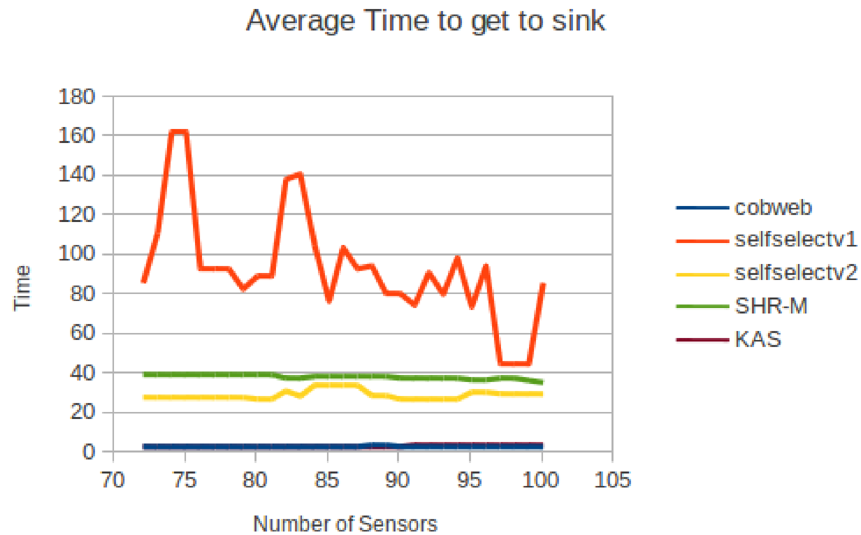

Figure 5.6: Number of Sensors vs Time to get to sink

The issue of data duplication can be easily seen by viewing the percentage of data received at the sink against the amount of data generated. Figure 5.7 indicates that the self selection V1 algorithm has large amounts of data duplication due to the equation allowing many sensors to select at the same time. The amount of time the simulation ran did not allow sensors to lose power permanently and therefore there was always a path to the sink. The cobweb and KAS algorithms correctly deliver 100% of the data to the sink whereas the self selection algorithms vary due to duplication and timing issues. The SHR-M algorithm has values under 100% due to the simplicity of the algorithm. When data had duplicated and both streams were approaching the back off timer for one stream was active when the other stream reached a sensor. This caused the sensor to think that another sensor had self selected and the data was incorrectly dropped.

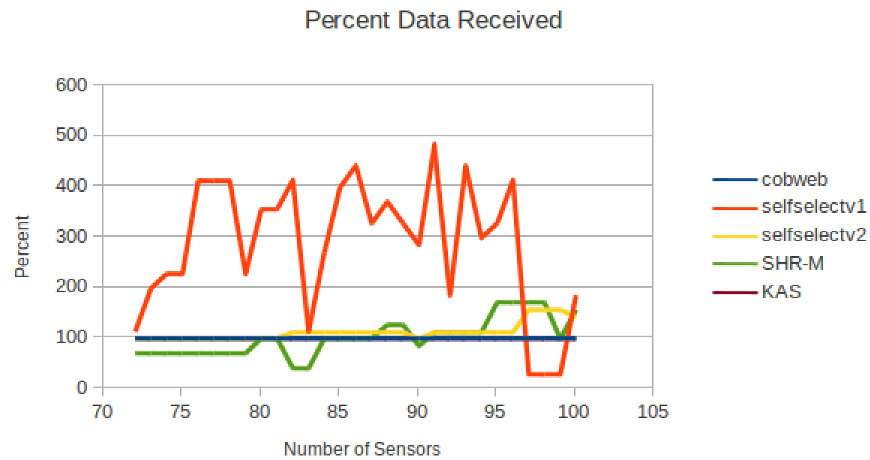

Figure 5.7: Percentage of data received at sink

The power of sensors was also monitored during these simulations using the same parameters and the average power remaining at the end of the simulation is displayed in Figure 5.8. The results show that the self selection v1 algorithm with an increased number of transmissions consistently has the least amount of power remaining at the end of the simulations. The SHR-M and KAS algorithms have similar power remaining due to the similar number of transmissions. The cobweb routing algorithm uses the least amount of power due to having the fewest transmission count as well as never waiting.

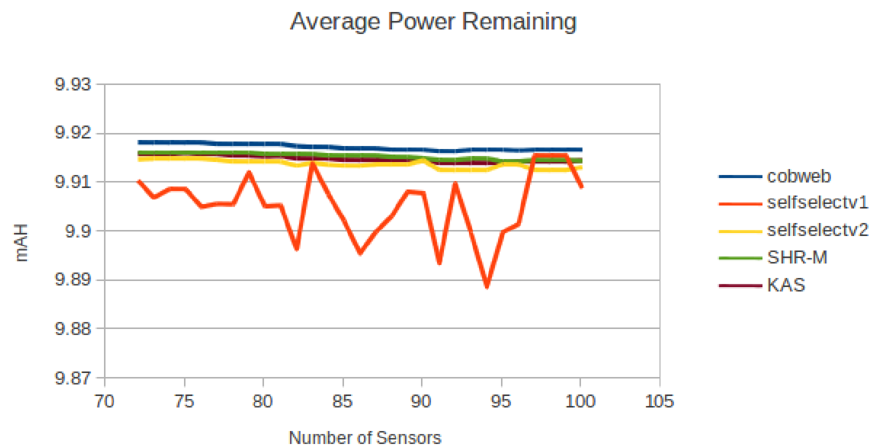

Figure 5.8: Average Power Remaining at End of Simulation

6. Future Improvements and Conclusion

6.1 Future Improvements

While the KAS algorithm performs very well in both speed and consistency, there are a few issues that could be addressed. In order to allow the algorithm to work with mobile sensors two features must be implemented. Whenever a sensor receives a data received message from an unknown sensor it must reply with a message indicating its own information. At that time, the sender can update the priority list incorporating the new sensor. Because this scheme could result in a situation where a sensor knows about every other sensor in the network a limit must be placed in order to preserve memory. This system could be as simple as a hard limit on the number of neighbors that are stored, or it could be a chatty system asking if a sensor is nearby before dropping it. In conjunction with these algorithms because the network would be constantly changing the sink would be required to send out broadcasts to inform the sensors to update the routing tables. A healing aspect could also be added to the KAS algorithm consistent with the SHR algorithm and would provide the same healing capabilities while keeping the property of having fewer transmissions overall.

Another feature that would allay the issue of delay when attempting to communicate with sensors that are asleep synchronization protocols such as the MAC protocols described in [37] or a simple synchronization protocol as used in SHR. A third and much simpler alternative that would allay this delay would be to change the priority of the sensor that was available allowing a learning process that would synchronize the sensors with a simple protocol.

A unique feature of the way the simulator is designed is that it is possible to generate the code for sequential simulation. Many of the features used in these algorithms have consistent structures in the code. All timers can be implemented the same way, knowledge of messages previously seen, and the order and type of messages passed between sensors can be generated with minimal loss in performance. This would allow any user to generate efficient code without any knowledge of how

the simulator itself works and could allow for extremely fast testing of algorithms. Optimistic simulation would be much more complicated and would more than likely require a programmer to generate the simulation, but the ability to rapidly test algorithms could be a very nice feature.

There could be a great deal more tests run to test the speed and efficiency of the simulator. Unfortunately, testing using ROSS' optimistic mode was unable to be successfully completed. But with optimistic mode large scale simulations using thousands of sensors should be feasible on a personal computer. ROSS also has the capability to be run on IBM Blue Gene super computers and could handle tens of thousands of processors generating and transmitting data.

6.2 Conclusion

By implementing several routing algorithms the efficiency and accuracy of the simulator has been demonstrated. The framework and design goals of the simulator allow algorithms to quickly and efficiently be implemented. By clearly identifying the basic aspects common to all algorithms from the specific requirements of each algorithm the simulator statistics can be consistently generated for use in comparing algorithms.

Several algorithms have been analyzed to demonstrate how the results of simulations can be used to visually display the effects for use in evaluating algorithms. A modification was made to the self selection family of algorithms to incorporate power into the data forwarding process. The results have been shown to increase the lifetime of sensors within the network in simulations.

Overall results show that the simulator that has been implemented is accurate and effective at running simulations and by using the examples of the models that have been implemented future algorithms can be implemented quickly.

LITERATURE CITED

[1] S. Tilak, N. B. Abu-Ghazaleh, and W. Heinzelman, "A taxonomy of wireless micro-sensor network models," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, pp. 28–36, April 2002.

[2] "Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks." IEEE Standard 802.15.4, 2006.

[3] "Tinyos home page." `http://www.tinyos.net/`, Sept. 2011.

[4] A. Woo, T. Tong, and D. Culler, "Taming the underlying challenges of reliable multihop routing in sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, (New York, NY, USA), pp. 14–27, ACM, 2003.

[5] J. Zhao and R. Govindan, "Understanding packet delivery performance in dense wireless sensor networks," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, (New York, NY, USA), pp. 1–13, ACM, 2003.

[6] G. Zhou, T. He, S. Krishnamurthy, and J. A. Stankovic, "Impact of radio irregularity on wireless sensor networks," in *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, (New York, NY, USA), pp. 125–138, ACM, 2004.

[7] M. Yarvis, W. Conner, L. Krishnamurthy, J. Chhabra, B. Elliott, and A. Mainwaring, "Real-world experiences with an interactive ad hoc sensor network," in *Parallel Processing Workshops, 2002. Proceedings. International Conference on*, pp. 143 – 151, 2002.

[8] J. Ahn and B. Krishnamachari, "Fundamental scaling laws for energy-efficient storage and querying in wireless sensor networks," in *Proceedings of the 7th ACM international symposium on Mobile ad hoc networking and computing*, MobiHoc '06, (New York, NY, USA), pp. 334–343, ACM, 2006.

[9] L. Eschenauer and V. D. Gligor, "A key-management scheme for distributed sensor networks," in *Proceedings of the 9th ACM conference on Computer and communications security*, CCS '02, (New York, NY, USA), pp. 41–47, ACM, 2002.

[10] G. Chen, J. Branch, and B. Szymanski, "Self-selective routing for wireless ad hoc networks," in *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005), IEEE International Conference on*, vol. 3, pp. 57 – 64 Vol. 3, aug. 2005.

[11] Z. Jizan, "A data aggregation and routing scheme of cobweb model in wireless sensor networks," in *Microwave, Antenna, Propagation and EMC Technologies for Wireless Communications, 2009 3rd IEEE International Symposium on*, pp. 323 –327, oct. 2009.

[12] Y. Wang, X. Fu, and L. Huang, "A storage node placement algorithm in wireless sensor networks," in *Frontier of Computer Science and Technology, 2009. FCST '09. Fourth International Conference on*, pp. 267 –271, dec. 2009.

[13] Z. Yu, B. Xiao, and S. Zhou, "Achieving optimal data storage position in wireless sensor networks," *Computer Communications*, vol. 33, no. 1, pp. 92 – 102, 2010.

[14] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran, "Ubiquitous access to distributed data in large-scale sensor networks through decentralized erasure codes," in *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, (Piscataway, NJ, USA), IEEE Press, 2005.

[15] S. Aly, M. Youssef, H. Darwish, and M. Zidan, "Distributed flooding-based storage algorithms for large-scale wireless sensor networks," in *Communications, 2009. ICC '09. IEEE International Conference on*, pp. 1 –5, june 2009.

[16] D. Niculescu and B. Nath, "Trajectory based forwarding and its applications," in *Proceedings of the 9th annual international conference on Mobile computing and networking*, MobiCom '03, (New York, NY, USA), pp. 260–272, ACM, 2003.

[17] S. Lee, B. Bhattacharjee, S. Banerjee, and B. Han, "A general framework for efficient geographic routing in wireless networks," *Computer Networks*, vol. 54, no. 5, pp. 844 – 861, 2010.

[18] J. Hornsberger and G. C. Shoja, "Geographic grid routing: designing for reliability in wireless sensor networks," in *Proceedings of the 2006 international conference on Wireless communications and mobile computing*, IWCMC '06, (New York, NY, USA), pp. 281–286, ACM, 2006.

[19] Y. Zhao, Y. Chen, and S. Ratnasamy, "Load balanced and efficient hierarchical data-centric storage in sensor networks," in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, pp. 560 –568, june 2008.

[20] K. Onodera and T. Miyazaki, "An autonomous algorithm for construction of energy-conscious communication tree in wireless sensor networks," in *Advanced Information Networking and Applications - Workshops, 2008. AINAW 2008. 22nd International Conference on*, pp. 898 –903, march 2008.

[21] S. Wan and Y. He, "Performance analysis of single-tree and split-tree approach in wireless sensor networks," in *Cyber-Enabled Distributed Computing and Knowledge Discovery, 2009. CyberC '09. International Conference on*, pp. 132 –135, oct. 2009.

[22] S. Hussain and O. Islam, "An energy efficient spanning tree based multi-hop routing in wireless sensor networks," in *Wireless Communications and Networking Conference, 2007.WCNC 2007. IEEE*, pp. 4383 –4388, march 2007.

[23] D. Oran, "Osi is-is intra-domain routing protocol." IETF RFC 1142, February 1990.

[24] F. Ye, H. Luo, J. Cheng, S. Lu, and L. Zhang, "A two-tier data dissemination model for large-scale wireless sensor networks," in *Proceedings of the 8th annual international conference on Mobile computing and networking*, MobiCom '02, (New York, NY, USA), pp. 148–159, ACM, 2002.

[25] M. Zeng and B. Xu, "A three-tiered topology control model for large scale scale wireless sensor networks," in *Control and Automation, 2007. IfCCA 2007. IEEE International Conference on*, pp. 3250 –3253, 30 2007-june 1 2007.

[26] G. Pei, M. Gerla, and X. Hong, "Lanmar: landmark routing for large scale wireless ad hoc networks with group mobility," in *Proceedings of the 1st ACM international symposium on Mobile ad hoc networking & computing*, MobiHoc '00, (Piscataway, NJ, USA), pp. 11–18, IEEE Press, 2000.

[27] P. F. Tsuchiya, "The landmark hierarchy: a new hierarchy for routing in very large networks," *SIGCOMM Comput. Commun. Rev.*, vol. 18, pp. 35–42, August 1988.

[28] Y. Sun, G. Fan, and S. Chen, "Spiral-based data dissemination in sensor networks," *Int. J. Ad Hoc Ubiquitous Comput.*, vol. 2, pp. 46–57, December 2007.

[29] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, (New York, NY, USA), pp. 81–94, ACM, 2004.

[30] H. Luo, G. Xing, M. Li, and X. Jia, "Dynamic multi-resolution data dissemination in storage-centric wireless sensor networks," in *Proceedings of the 10th ACM Symposium on Modeling, analysis, and simulation of wireless and mobile systems*, MSWiM '07, (New York, NY, USA), pp. 78–85, ACM, 2007.

[31] P. Skraba, Q. Fang, A. Nguyen, and L. Guibas, "Sweeps over wireless sensor networks," in *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, (New York, NY, USA), pp. 143–151, ACM, 2006.

[32] D. Braginsky and D. Estrin, "Rumor routing algorthim for sensor networks," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, (New York, NY, USA), pp. 22–31, ACM, 2002.

[33] G. Di Caro, F. Ducatelle, and L. M. Gambardella, "Anthocnet: an adaptive nature-inspired algorithm for routing in mobile ad hoc networks," *European Transactions on Telecommunications*, vol. 16, no. 5, pp. 443–455, 2005.

[34] J. Liu, F. Zhao, and D. Petrovic, "Information-directed routing in ad hoc sensor networks," in *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, (New York, NY, USA), pp. 88–97, ACM, 2003.

[35] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, (New York, NY, USA), pp. 56–67, ACM, 2000.

[36] W. Ye, J. Heidemann, and D. Estrin, "Medium access control with coordinated adaptive sleeping for wireless sensor networks," *Networking, IEEE/ACM Transactions on*, vol. 12, pp. 493 – 506, june 2004.

[37] I. Demirkol, C. Ersoy, and F. Alagoz, "Mac protocols for wireless sensor networks: a survey," *Communications Magazine, IEEE*, vol. 44, pp. 115 – 121, april 2006.

[38] G. Chen, J. Branch, and B. Szymanski, "A self-selection technique for flooding and routing in wireless ad-hoc networks," *Journal of Network and Systems Management*, vol. 14, pp. 359–380, 2006. 10.1007/s10922-006-9036-7.

[39] T. A. Babbitt, C. Morrell, B. K. Szymanski, and J. W. Branch, "Self-selecting reliable paths for wireless sensor network routing," *Computer Communications*, vol. 31, no. 16, pp. 3799 – 3809, 2008. ¡ce:title¿Performance Evaluation of Communication Networks (SPECTS 2007)¡/ce:title¿.

[40] D. Ganesan, R. Govindan, S. Shenker, and D. Estrin, "Highly-resilient, energy-efficient multipath routing in wireless sensor networks," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 5, pp. 11–25, October 2001.

[41] "Crossbow technology inc.." `http://www.xbow.com/`, Sept. 2011.

APPENDIX A
Source Code

A.1 Description

The source code is split into two parts; There is the main code in the top directory that are common to all algorithm simulations and there are sub folders containing the algorithm implementations.

The base directory contains 4 files: wsn.c definitions.h, utils.h, and CMakeLists.txt.

- wsn.c is the main function of the program and handles the initialization and conclusion of ROSS as well as the main event handler. The main event handler calls functions within each algorithm according to the events required.
- definitions.h includes common definitions used throughout all of the algorithms.
- utils.h contains functions that are common to more than one algorithm.
- CMakeLists.txt is a file for use with CMake. ROSS uses CMake to build the source code and it has been included as instruction into how to easily change between algorithms. The way the code has been built, to use an algorithm merely requires a preprocessor definition of the algorithm's name to be used. Within the source code, preprocessor directives are used to include the functions and files required for that algorithm.It is only necessary to uncomment one algorithm in this file to select the algorithm desired.

Within each algorithm directory are 3 files: messageUtils.h, wsn.h, and algorithm.h (where the name of the algorithm replaces 'algorithm').

- messageUtils.h contains all of the functions to create events (aka messages) as well as to send them to other sensors.

- wsn.h defines the structures used in the algorithm such as the information a sensor has and the information stored in a message. This file may also contain definitions that are specific to this algorithm.

- algorithm.h contains the implementations of the functions called by wsn.c on the top level. This is truly what allows the algorithms to have different implementations and can be created very easily.

A.2 Top Level Source Files

Listing A.1: Source Code

```
1  /**
2   * @file wsn.c
3   * @brief Functions implementing ross functions.
4   * @author Thomas Reale
5   * @date 12-2011
6   * @version 1.0
7   *
8   * Defines the base structure for the functinos that are required by
9   * ROSS to run the simulation. Initializes the network by giving sensors
10  * a location in a grid.
11  *
12  * Events should be defined in another header file according to the routing
13  * algorithm. Routing algorithms are used at compile time based on the
14  * command line definitions in the CMakeLists file.
15  *
16  * Displays results of simulation at completion
17  */
18
19 // Build arguments determine what type of algthm is run
20 #include "definitions.h"
21
22 #ifdef DIJKSTRA
23 #include "dijkstra/wsn.h"
24 #include "dijkstra/dijkstra.h"
25 #elif SELFSELECT
26 #include "selfselect/wsn.h"
27 #include "selfselect/selfselect.h"
28 #elif COBWEB
29 #include "cobweb/wsn.h"
30 #include "cobweb/cobweb.h"
31 #elif ASSISTED
32 #include "assisted/wsn.h"
33 #include "assisted/assisted.h"
34 #else
35 #error Need to specify algorithm (DIJKSTRA,SELFSELECT,COBWEB,ASSISTED)
36 #endif
```

```
37
38 #include "math.h"
39 #include "utils.h"
40
41
42 /**
43  * Mapping of global id's to logical processors.
44  * Uses a linear mapping
45  *
46  * @param gid The sensor's global ID being mapped
47  */
48   tw_peid
49 mapping(tw_lpid gid)
50 {
51         return (tw_peid) gid / g_tw_nlp;
52 }
53
54 /**
55  * The init function initializes the state of a single sensor.
56  *
57  * @param s The sensor state for a given sensor.
58  * @param lp The lp associated with the sensor
59  */
60 void
61 init(sensor_state *s, tw_lp * lp)
62 {
63   tw_event *e;
64   sensor_message *m;
65   Sensor *thisSensor = (Sensor *) malloc(sizeof(Sensor));
66
67   initializeVariables(s);
68
69   if (lp->gid == 0) {
70     s->location.x = 0;
71     s->location.y = 0;
72   } else if (lp->gid == 1) {
73     s->location.x = 10;
74     s->location.y = 10;
75   } else {
76     do {
77       s->location.x = tw_rand_integer(lp->rng, 0, grid_width);
78       s->location.y = tw_rand_integer(lp->rng, 0, grid_width);
79     } while (checkUnique(s->location.x, s->location.y, lp->gid) == 0);
80   }
81   // Set all information about sinks to UNKNOWN
82 #ifndef COBWEB
83   int i = 0;
84   int j = 0;
85   for (i = 0; i < MAX_SINKS; ++i) {
86     for (j = 0; j < MAX_PATHS_PER_SINK; ++j) {
87       s->sink_path[i][j] = UNKNOWN;
88       s->sink_dist[i][j] = UNKNOWN;
```

```
89        }
90        s->sink_gid[i] = UNKNOWN;
91      }
92  #endif
93      /** saves this sensor's location and ID for communication with others */
94      thisSensor->location.x = s->location.x;
95      thisSensor->location.y = s->location.y;
96      thisSensor->gid = lp->gid;
97
98      sensors_list[lp->gid] = thisSensor;
99
100     // Tell the sensors to initialize. Discover nearby neighbors
101     e = tw_event_new(lp->gid, 0, lp);
102     m = tw_event_data(e);
103     m->type = INIT;
104     tw_event_send(e);
105
106 }
107
108 /**
109  * The action to be taken when an event occurs at this sensor.
110  *
111  * The actions are determined by the routing algorithm. The functions
112  * for each message type should be defined in a header file with the associated
113  * algorithm.
114  *
115  * The algorithm is selected using a command line definition.
116  *
117  * @param s The state of the sensor
118  * @param bf The time warp bit field used to indicate actions taken.
119  * @param msg The msg associated with this event.
120  * @param lp The logical processor associated with this event.
121  */
122 void
123 event_handler(sensor_state * s, tw_bf * bf, sensor_message * msg, tw_lp * lp)
124 {
125
126     if (msg->type != INIT && s->power <= 0) {
127       if (s->state != DEAD) {
128         printf("%li Died\n", lp->gid);
129         s->state = DEAD;
130       }
131       return;
132     }
133
134     switch(msg->type)
135       {
136
137       case INIT:
138         sensor_init(s, bf, msg, lp);
139         break;
140       case SLEEP:
```

```
141 #ifdef VERBOSE
142         if (WHO_GEN_DATA)
143           printf("%li_SLEEP\n", lp->gid);
144 #endif
145         sleep_event(s, bf, msg, lp);
146         break;
147       case AWAKE:
148 #ifdef VERBOSE
149         if (WHO_GEN_DATA)
150           printf("%li_AWAKE\n", lp->gid);
151 #endif
152         awake_event(s, bf, msg, lp);
153         break;
154       case SINK_LOC:
155         sink_loc_event(s, bf, msg, lp);
156         break;
157       case SINK_LOST:
158         sink_lost_event(s, bf, msg, lp);
159         break;
160       case DATA_GEN:
161         total_data_generated += data_size;
162         if (s->data_size >= sensor_storage) {
163           total_data_overflow += data_size;
164         } else {
165           data_gen_event(s, bf, msg, lp);
166         }
167         break;
168       case DATA_RECEIVED:
169 #ifdef VERBOSE2
170         printf("%li_RECEIVED_DATA:state:%d\n", lp->gid, s->state);
171 #endif
172           data_received_event(s, bf, msg, lp);
173         break;
174 #ifdef DIJKSTRA
175       case DATA_SEND:
176           data_send_event(s, bf, msg, lp);
177         break;
178 #endif
179 #ifndef DIJKSTRA
180 #ifndef COBWEB
181       case WAIT:
182 #ifdef VERBOSE
183         if (WHO_GEN_DATA)
184           printf("%li_WAIT_EXPIRED_%G\n", lp->gid, (double)tw_now(lp));
185 #endif
186           wait_event(s, bf, msg, lp);
187         break;
188 #endif
189 #endif
190 #ifdef SELFSELECT
191       case DATA_ACK:
192 #ifdef VERBOSE
```

```
193         printf("%li ACK\n", lp->gid);
194 #endif
195         data_ack_event(s, bf, msg, lp);
196       break;
197 #endif
198 #ifdef COBWEB
199     case TRANSMIT_COMPLETE:
200 #ifdef VERBOSE
201       printf("%li TRANSMIT_COMPLETE\n", lp->gid);
202 #endif
203       transmit_complete_event(s, bf, msg, lp);
204       break;
205 #endif
206 #ifdef ASSISTED
207     case DATA_EXPIRED:
208       expired_event(s, bf, msg, lp);
209       break;
210     case DATA_FORWARD:
211         data_forward_event(s, bf, msg, lp);
212       break;
213     case DATA_ACK:
214         data_ack_event(s, bf, msg, lp);
215       break;
216 #endif
217     default:
218       printf("ERROR: Unknown event %d\n", msg->type);
219       break;
220     }
221 }
222
223 /**
224  * The Reverse Compiler event handler
225  *
226  * Reverses any operation that was completed in the handler.
227  * An AWAKE message indicates the start of an AWAKE cycle.
228  * A  SLEEP message indicates the start of a SLEEP cycle.
229  *
230  * @param s The state of the sensor.
231  * @param bf The time warp bit field indicating logical paths taken.
232  * @param msg The message that is being reversed.
233  * @param lp The logical processor associated with this sensor.
234  */
235 void
236 rc_event_handler(sensor_state * s, tw_bf * bf, sensor_message * msg, tw_lp * lp)
237 {
238   switch(msg->type)
239   {
240     case INIT:
241       sensor_init_rc(s, bf, msg, lp);
242       break;
243     case SLEEP:
244       sleep_event_rc(s, bf, msg, lp);
```

```
245         break;
246       case AWAKE:
247         awake_event_rc(s, bf, msg, lp);
248         break;
249       case SINK_LOC:
250         sink_loc_event_rc(s, bf, msg, lp);
251         break;
252       case SINK_LOST:
253         sink_lost_event_rc(s, bf, msg, lp);
254         break;
255       case DATA_GEN:
256         total_data_generated -= data_size;
257         data_gen_event_rc(s, bf, msg, lp);
258         break;
259       case DATA_RECEIVED:
260 #ifdef VERBOSE2
261         printf("%li RECEIVED DATA\n", lp->gid);
262 #endif
263         data_received_event_rc(s, bf, msg, lp);
264         break;
265 #ifdef COBWEB
266       case TRANSMIT_COMPLETE:
267         transmit_complete_event_rc(s, bf, msg, lp);
268         break;
269 #endif
270 #ifdef DIJKSTRA
271       case DATA_SEND:
272         data_send_event_rc(s, bf, msg, lp);
273         break;
274 #endif
275 #ifndef DIJKSTRA
276 #ifndef COBWEB
277       case WAIT:
278         wait_event_rc(s, bf, msg, lp);
279         break;
280 #endif
281 #endif
282 #ifdef SELFSELECT
283       case DATA_ACK:
284 #ifdef VERBOSE
285         printf("%li ACK\n", lp->gid);
286 #endif
287         data_ack_event_rc(s, bf, msg, lp);
288         break;
289 #endif
290 #ifdef ASSISTED
291       case DATA_EXPIRED:
292         expired_event_rc(s, bf, msg, lp);
293         break;
294       case DATA_FORWARD:
295         data_forward_event_rc(s, bf, msg, lp);
296         break;
```

```
297       case DATA_ACK:
298         data_ack_event_rc(s, bf, msg, lp);
299         break;
300 #endif
301       default:
302         break;
303     }
304 }
305
306 /**
307  * A function that is performed after all other operations have completed.
308  * Used for gathering statistics.
309  *
310  * @param s The final state of the sensor.
311  * @param lp The logical processor associated with the sensor.
312  */
313 void
314 final(sensor_state * s, tw_lp * lp)
315 {
316
317   // Ignore awake/asleep for sink. Assume always on.
318   if (!(WHO_IS_SINK)) {
319     awake_time_avg += ((s->total_awake_time /
320                         (double) s->times_awake) / (gnum_sensors -1));
321     asleep_time_avg += ((s->total_asleep_time /
322                         (double) s->times_asleep) / (gnum_sensors -1));
323
324     average_power_remaining += (s->power / (gnum_sensors -1));
325     if (s->power <= 0) {
326       number_dead_sensors++;
327     }
328   }
329   transmit_time_avg += (s->total_transmit_time / gnum_sensors);
330
331   if (s->total_transmit_time > transmit_time_max) {
332     transmit_time_max = s->total_transmit_time;
333   }
334   num_msg_avg += (s->messages_sent / gnum_sensors);
335   total_msg_count += s->messages_sent;
336
337   if (s->messages_sent > max_msg_single) {
338     max_msg_single = s->messages_sent;
339   }
340
341   printf("GID:%li:", lp->gid);
342
343
344   int i = 0;
345   printf("nearby:");
346   for (i = 0; i < s->num_nearby; ++i) {
347 #ifdef COBWEB
348     printf("%li,", s->nearby_Sensors[i].gid);
```

```
349 #else
350       printf("%li,", s->nearby_Sensors[i]);
351 #endif
352     }
353     printf("\n");
354     free(s->nearby_Sensors);
355 }
356
357 /**
358  * Defines the functions that will be used for the ROSS operations
359  */
360 tw_lptype sensor_lps[] =
361 {
362         {
363                 (init_f) init,
364                 (event_f) event_handler,
365                 (revent_f) rc_event_handler,
366                 (final_f) final,
367                 (map_f) mapping,
368                 sizeof(sensor_state),
369         },
370         {0},
371 };
372
373
374 /**
375  * Defines any extra parameters used for the application
376  */
377 const tw_optdef app_opt [] =
378 {
379             TWOPT_GROUP("Wireless Sensor Network Model"),
380             TWOPT_UINT("nsensors", gnum_sensors,
381         "The number of sensors in the simulation."),
382     TWOPT_UINT("width", grid_width,
383         "The width of the square defining the region sensors are distributed."),
384     TWOPT_UINT("range", sensor_range,
385         "The range that a sensor can communicate."),
386     TWOPT_UINT("power", sensor_power,
387         "The initial power that a sensor contains in mAH."),
388     TWOPT_UINT("tsleep", gsleep_time,
389         "The length of time a sensor sleeps in a single cycle."),
390     TWOPT_UINT("tawake", gawake_time,
391         "The length of time a sensor is awake in a single cycle."),
392     TWOPT_UINT("delay", transmit_delay_tenths,
393         "A delay added to all communicationsi in 1/10ths of seconds."
394         "Also used as transmission time for small packets and standard deviation"
395         " in normal distribution to add random factor."),
396     TWOPT_UINT("data_size", data_size,
397         "The amount of data generated at a time in bits."),
398     TWOPT_UINT("gen_rate", gdata_gen_rate,
399         "The rate at which data is generated."),
400     TWOPT_UINT("transmit_rate", gtransmit_rate,
```

```
401         "The rate at which the sensors transmit data in bits/sec"),
402
403    TWOPT_END()
404 };
405
406 /**
407  * The main function called when the program is run.
408  * Processes all operation global to the program.
409  *
410  * @param argc The number of arguments.
411  * @param argv The arguments
412  * @param env The environment variables.
413  */
414 int
415 main(int argc, char **argv, char **env)
416 {
417
418
419         int i;
420
421   // Add the command line options
422         tw_opt_add(app_opt);
423
424   // Initialize ross arguments
425         tw_init(&argc, &argv);
426
427   sensors_list = (Sensor **) malloc(sizeof(Sensor *) * gnum_sensors);
428   transmission_delay = (double)transmit_delay_tenths/10;
429
430   // Set up the memory for the nodes
431         nlp_per_pe = gnum_sensors/(tw_nnodes() * g_tw_npe);
432         g_tw_events_per_pe = (nlp_per_pe / g_tw_npe);
433
434         tw_define_lps(nlp_per_pe, sizeof(sensor_message), 0);
435
436         for(i = 0; i < g_tw_nlp; i++)
437                 tw_lp_settype(i, &sensor_lps[0]);
438
439   // run the simulation
440         tw_run();
441
442   long double time_msg_avg = transmission_path_time_total / transmission_recv_ct;
443
444   // Display all information specific to this application.
445         if(tw_ismaster())
446         {
447     printf("\t%-50s %11lld\n", "Number of Sensors",
448                         nlp_per_pe * g_tw_npe * tw_nnodes());
449     printf("\t%-50s %11d\n", "Grid Width", grid_width);
450     printf("\t%-50s %11d\n", "Sensor Range", sensor_range);
451     printf("\t%-50s %11d\n", "Data Gen Rate", gdata_gen_rate);
452     printf("\t%-50s %11d\n", "Data Transmit Rate", gtransmit_rate);
```

```
453       printf("\t%-50s %11d\n", "Data Size Generated", data_size);
454       printf("\t%-50s %11.2G\n", "Transmission Delay", transmission_delay);
455       printf("\t%-50s %11.2G\n", "Transmit Standard Deviation", std);
456
457                   printf("\nWireless Sensor Network Model Statistics:\n");
458       printf("\nTiming Statistics:\n");
459                   printf("\t%-50s %11.4lf\n", "Average Awake Time", awake_time_avg);
460       printf("\t%-50s %11.4lf\n", "Average Asleep Time", asleep_time_avg);
461       printf("\t%-50s %11.4lf\n", "Average Sensor Transmit Time", transmit_time_avg);
462       printf("\t%-50s %11.4lf\n", "Max Sensor Transmit Time", transmit_time_max);
463       printf("\t%-50s %11.4Lf\n", "Average time to get to sink", time_msg_avg);
464       printf("\nNetwork Transmission Statistics:\n");
465       printf("\t%-50s %11.4f\n", "Total transmission count", total_msg_count);
466       printf("\t%-50s %11.4f\n", "Average transmission count", num_msg_avg);
467       printf("\t%-50s %11.4f\n", "Max transmissions single sensor", max_msg_single);
468       printf("\nData Routing Statistics:\n");
469       printf("\t%-50s %11.4f\n", "Total data generated", total_data_generated);
470       printf("\t%-50s %11.4f\n", "Data Sink Received", data_sink_received);
471       printf("\t%-50s %11.4f\n", "Data lost due to overflow", total_data_overflow);
472       printf("\nPower Statistics:\n");
473       printf("\t%-50s %11.4f\n", "Average Power Remaining", average_power_remaining);
474       printf("\t%-50s %11.4d\n", "Number of Dead Sensors", number_dead_sensors);
475       printf("\n\n\n");
476       printf("\n\nLocations of wireless sensors:\n");
477       printf("\nsize:%d\n", gnum_sensors);
478       printf("width:%d\n", grid_width);
479       for (i = 0; i < gnum_sensors; ++i) {
480         printf("GID:%d:location:%li,%li\n", i, sensors_list[i]->location.x, sensors_lis
481       }
482
483     }
484
485     // Wrap up stuff
486           tw_end();
487
488           return 0;
489   }
```

Listing A.2: Source Code

```
1   /**
2    * @file utils.h
3    * @brief Functions implementing ross functions.
4    * @author Thomas Reale
5    * @date 12-2011
6    * @version 1.0
7    *
8    * Defines functions that are useful in multiple simulations.
9    */
10
11  #ifndef WSN_UTILS_H
12  #define WSN_UTILS_H
13
```

```
14 #ifdef DIJKSTRA
15 #include "dijkstra/wsn.h"
16 #elif SELFSELECT
17 #include "selfselect/wsn.h"
18 #elif COBWEB
19 #include "cobweb/wsn.h"
20 #endif
21 #include "math.h"
22
23 /**
24  * Calculates the power loss in mAH for a given amount of time and loss power usage.
25  *
26  * @param time The amount of seconds spent.
27  * @param rate The rate at which power is used (uA)
28  *
29  * @return The power loss in mAH
30  */
31 double power_loss(double time, double rate) {
32   double hours = time/3600;
33   double loss = rate*hours/1000;
34   return loss;
35 }
36
37 /**
38  * Determines if this point given is unique among all sensors
39  *
40  * @param x The x location to check.
41  * @param y The y location to check.
42  * @param sensorNum The new sensor number to know how many nodes are in the grid.
43  *
44  * @return 0 if not unique, 1 if unique.
45  */
46 int checkUnique(int x, int y, int sensorNum) {
47   int i = 0;
48   for (i = 0; i < sensorNum; ++i) {
49     if (sensors_list[i]->location.x == x && sensors_list[i]->location.y == y) {
50       return 0;
51     }
52   }
53   return 1;
54 }
55
56 /**
57  * Copies all the message data from the source to the destination.
58  *
59  * @param src the source of the information.
60  * @param dst The destination of the information.
61  */
62 void copyMessageData(sensor_message *src, sensor_message *dst) {
63   if (src != NULL && dst != NULL) {
64     dst->type = src->type;
65     dst->data_size = src->data_size;
```

```
66      dst->num_hops = src->num_hops;
67      dst->last_gid = src->last_gid;
68      dst->start_time = src->start_time;
69      dst->last_power = src->last_power;
70
71  #ifndef COBWEB
72      dst->dest_gid = src->dest_gid;
73      dst->source_gid = src->source_gid;
74      dst->last_last_gid = src->last_last_gid;
75      dst->nonce = src->nonce;
76      dst->sink_gid = src->sink_gid;
77  #endif
78    }
79  }
80  /**
81   * Returns the larger of the two values given.
82   *
83   * @param val1 The first value
84   * @param val2 The second value
85   *
86   * @return The larger of the two given values
87   */
88  double maxVal(double val1, double val2) {
89    return (val1 > val2 ? val1 : val2);
90  }
91
92  /**
93   * Returns the distance between the two locations.
94   *
95   * @param loc1 The first location.
96   * @param loc2 The secodn location
97   *
98   * @return The distance between locations
99   */
100 double getDistance(Location loc1, Location loc2) {
101   double a2 = pow((double)(loc2.x-loc1.x),2);
102   double b2 = pow((double)(loc2.y-loc1.y),2);
103
104   double c2 = a2+b2;
105   double c = sqrt(c2);
106   return c;
107 }
108
109 /**
110  * Finds sensors that are within sensor_range units of the given sensor.
111  *
112  * @param s The sensor to find neighbors for.
113  * @param myGid The gid of this sensor.
114  *
115  */
116 void findNearbySensors(sensor_state *s, long myGid) {
117   // Initially allocate the number to be the expected number
```

```
118     // of sensors (by density * MAX_NEARBY)
119     float nearbyEstimate = gnum_sensors/pow(grid_width, 2);
120     if (nearbyEstimate == 0) {
121       nearbyEstimate += 1;
122     }
123     // Accomodate for the range that a sensor can reach
124     nearbyEstimate *= 8*sensor_range;
125     // Accomodate for randomness
126     nearbyEstimate *= 2;
127
128  #ifdef COBWEB
129     Neighbor *nearby = malloc(sizeof(Neighbor) * (int)nearbyEstimate);
130  #else
131     long *nearby = malloc(sizeof(long) * (int)nearbyEstimate);
132  #endif
133     int numNearby = 0;
134
135     int i = 0;
136     for (i = 0; i < gnum_sensors; ++i) {
137       double dist = getDistance(s->location, sensors_list[i]->location);
138       if (dist <= sensor_range && numNearby < nearbyEstimate) {
139         // Don't include self
140         if (sensors_list[i]->gid != myGid) {
141  #ifdef COBWEB
142           nearby[numNearby].gid = sensors_list[i]->gid;
143  #else
144           nearby[numNearby] = sensors_list[i]->gid;
145  #endif
146           numNearby++;
147         }
148       } else if (numNearby == nearbyEstimate) {
149         // Resize the nearby array
150         printf("ERROR: Sensor %li didn't allocate enough space for nearby sensors. Nu
151         break;
152       }
153     }
154
155     s->num_nearby = numNearby;
156     s->nearby_Sensors = nearby;
157  }
158
159  #ifndef COBWEB
160  /**
161   * Checks to see if a sink has already been learned
162   *
163   * @param s The sensor to check.
164   * @param sink_gid The sink to check.
165   *
166   * @return The location in the sink_gid array of the sink if it exists
167   *         otherwise -1.
168   */
169  int check_for_sink(sensor_state *s, double sink_gid) {
```

```
170   int returnValue = -1;
171
172   int i = 0;
173   for (i = 0; i < MAX_SINKS; ++i) {
174     if (s->sink_gid[i] == sink_gid) {
175       returnValue = i;
176       break;
177     }
178   }
179   return returnValue;
180 }
181
182 /**
183  * Checks all sink paths to determine if the path is being updated
184  *
185  * @param s The sensor to check.
186  * @param sinkId The ID of the sink in the sink_gid array to check the paths for.
187  * @param sink_path The path to check if being updated.
188  *
189  * @return The index of the path to be updated if needed.
190  *         -1 if no update needed.
191  */
192 int check_path_update(sensor_state *s, int sinkId, double sink_path) {
193   int returnValue = -1;
194
195   int i = 0;
196   for (i = 0; i < MAX_PATHS_PER_SINK; ++i) {
197     if (s->sink_path[sinkId][i] == sink_path) {
198       returnValue = i;
199       break;
200     }
201   }
202   return returnValue;
203 }
204
205 /**
206  * Creates room for a new path to an old sink to be inserted into the given location
207  * in the given sensor. UNKNOWN values are placed in the freed locatino.
208  * The worst path is removed if necessary.
209  *
210  * @param s The sensor we are making room in.
211  * @param sink The sink array location.
212  * @param dist The dist array location.
213  */
214 void make_room_path(sensor_state *s, int sink, int dist) {
215   int j = 0;
216
217   for (j = MAX_PATHS_PER_SINK-2; j >= dist; --j) {
218     s->sink_path[sink][j+1] = s->sink_path[sink][j];
219     s->sink_dist[sink][j+1] = s->sink_dist[sink][j];
220   }
221   s->sink_path[sink][j+1] = UNKNOWN;
```

```
222     s->sink_dist[sink][j+1] = UNKNOWN;
223   }
224
225   /**
226    * Creates room for a new sink in the location given.
227    * UNKNOWN values are placed in the cleared location.
228    *
229    * @param s The sensor to update
230    * @param sinkLoc The location of a sink to clear.
231    */
232   void make_room_sink(sensor_state *s, int sinkLoc) {
233     int i = 0;
234     int j = 0;
235
236     for (i = MAX_SINKS-2; i >= sinkLoc; --i) {
237
238       for (j = 0; j < MAX_PATHS_PER_SINK; ++j) {
239         s->sink_path[i+1][j] = s->sink_path[i][j];
240         s->sink_dist[i+1][j] = s->sink_dist[i][j];
241       }
242       s->sink_gid[i+1] = s->sink_gid[i];
243     }
244     s->sink_gid[i+1] = UNKNOWN;
245     for (j = 0; j < MAX_PATHS_PER_SINK; ++j) {
246       s->sink_path[i+1][j] = UNKNOWN;
247       s->sink_dist[i+1][j] = UNKNOWN;
248     }
249   }
250
251   /**
252    * Swaps the data in the paths within the given sink.
253    *
254    * @param s The sensor where sinks are being swapped.
255    * @param sink The sink this path is associated with.
256    * @param path1 A path to swap
257    * @param path2 A path to swap
258    */
259   void swap_paths(sensor_state *s, int sink, int path1, int path2) {
260     int temp_path;
261     int temp_dist;
262  #ifdef ASSISTED
263     double temp_cost;
264  #endif
265     temp_path = s->sink_path[sink][path1];
266     temp_dist = s->sink_dist[sink][path1];
267  #ifdef ASSISTED
268     temp_cost = s->sink_cost[sink][path1];
269  #endif
270
271     s->sink_path[sink][path1] = s->sink_path[sink][path2];
272     s->sink_dist[sink][path1] = s->sink_dist[sink][path2];
273  #ifdef ASSISTED
```

```
274     s->sink_cost[sink][path1] = s->sink_cost[sink][path2];
275  #endif
276
277     s->sink_path[sink][path2] = temp_path;
278     s->sink_dist[sink][path2] = temp_dist;
279  #ifdef ASSISTED
280     s->sink_cost[sink][path2] = temp_cost;
281  #endif
282  }
283  /**
284   * Swaps the data for two sinks within the given sensor.
285   *
286   * @param s The sensor where sinks are being swapped.
287   * @param sinkLoc1 location of a sink in s->sink_gid.
288   * @param sinkLoc2 location of a sink in s->sink_gid
289   */
290  void swap_sinks(sensor_state *s, int sinkLoc1, int sinkLoc2) {
291     double temp_path[MAX_PATHS_PER_SINK];
292     int temp_dist[MAX_PATHS_PER_SINK];
293     double temp_gid;
294
295     int i = 0;
296     for (i = 0; i < MAX_PATHS_PER_SINK; ++i) {
297        temp_path[i] = s->sink_path[sinkLoc1][i];
298        temp_dist[i] = s->sink_dist[sinkLoc1][i];
299     }
300     temp_gid = s->sink_gid[sinkLoc1];
301
302     for (i = 0; i < MAX_PATHS_PER_SINK; ++i) {
303        s->sink_path[sinkLoc1][i] = s->sink_path[sinkLoc2][i];
304        s->sink_dist[sinkLoc1][i] = s->sink_dist[sinkLoc2][i];
305     }
306     s->sink_gid[sinkLoc1] = s->sink_gid[sinkLoc2];
307
308     for (i = 0; i < MAX_PATHS_PER_SINK; ++i) {
309        s->sink_path[sinkLoc2][i] = temp_path[i];
310        s->sink_dist[sinkLoc2][i] = temp_dist[i];
311     }
312     s->sink_gid[sinkLoc2] = temp_gid;
313  }
314
315  /**
316   * Adds a sink to the given sensor's list of sinks.
317   * If there is no more space, replaces the sink with the worst priority.
318   * If equal distance to the worst one stored, and no extra space
319   * keeps the sink already there.
320   * Arrays are ordered according to distance and power.
321   *
322   * @param s The sensor who learned about the sink.
323   * @param sink_gid The gid of the sink.
324   * @param sink_dist The distance to the sink.
325   * @param sink_path The sensor to send data to for that sink.
```

```
326    * @param path_power For the KASS algorithm the power remaining in the path sensor.
327    *
328    * @return A 1 if the sink was added/modified or a 0 if no updates were made.
329    */
330   int add_sink(sensor_state *s, double sink_gid, double sink_path, int sink_dist,
331                int path_power) {
332     int returnValue = 0;
333     int i = 0;
334 
335  #ifdef ASSISTED
336     double cost = (double)sink_dist + log((path_power/sensor_power));
337  #endif
338     int sinkLocation = check_for_sink(s, sink_gid);
339     if (sinkLocation != -1) {
340       // If paths already exist for this sink update them.
341 
342       // if path just being updated update it
343       int existingPath = check_path_update(s, sinkLocation, sink_path);
344       if (existingPath != -1) {
345         if (s->sink_dist[sinkLocation][existingPath] > sink_dist) {
346           s->sink_dist[sinkLocation][existingPath] = sink_dist;
347  #ifdef ASSISTED
348           s->sink_cost[sinkLocation][existingPath] = cost;
349  #endif
350           returnValue = 1;
351         }
352       } else {
353         // else the path is new
354         // Find where to place this path
355         for (i = 0; i < MAX_PATHS_PER_SINK; ++i) {
356           if (s->sink_dist[sinkLocation][i] > sink_dist) {
357             make_room_path(s, sinkLocation, i);
358           }
359           if (s->sink_dist[sinkLocation][i] == UNKNOWN) {
360             s->sink_dist[sinkLocation][i] = sink_dist;
361             s->sink_path[sinkLocation][i] = sink_path;
362  #ifdef ASSISTED
363           s->sink_cost[sinkLocation][existingPath] = cost;
364  #endif
365             returnValue = 1;
366             break;
367           }
368         }
369       }
370     } else {
371       // The sink is newly discovered.
372       // Find where to place the sink. Place is determined by best distance to sink.
373       for (i = 0; i < MAX_SINKS; ++i) {
374  #ifdef ASSISTED
375         if (s->sink_cost[i][0] > cost) {
376           make_room_sink(s, i);
377         }
```

```
378 #else
379         if (s->sink_dist[i][0] > sink_dist) {
380           make_room_sink(s, i);
381         }
382 #endif
383         if (s->sink_gid[i] == UNKNOWN) {
384           s->sink_gid[i] = sink_gid;
385           s->sink_dist[i][0] = sink_dist;
386           s->sink_path[i][0] = sink_path;
387 #ifdef ASSISTED
388           s->sink_cost[i][0] = cost;
389 #endif
390           sinkLocation = i;
391           returnValue = 1;
392           break;
393         }
394       }
395     }
396
397     // Now need to make sure that the order is still maintained for sinks
398     for (i = sinkLocation-1; i >= 0; --i) {
399 #ifdef ASSISTED
400       if (s->sink_cost[i][0] > s->sink_cost[i+1][0]) {
401 #else
402       if (s->sink_dist[i][0] > s->sink_dist[i+1][0]) {
403 #endif
404           swap_sinks(s, i, i+1);
405       } else {
406         break;
407       }
408     }
409     return returnValue;
410 }
411
412
413 /**
414  * Updates the priorities for a single path.
415  * For the assisted algorithm the power is incorporated into the value.
416  * Otherwise function does nothing.
417  *
418  * @param s The sensor to remove the sink from.
419  * @param sink_gid The gid of the sink to remove.
420  * @param sink_path The gid of the sensor in the path to the sik.
421  * @param sink_dist The distance to the sink.
422  * @param path_power the power remaining on the sink_path sensor.
423  *
424  * @return 0 if the sink was not found. 1 if found and removed.
425  */
426 int update_cost(sensor_state *s, double sink_gid, double sink_path, int sink_dist,
427                 int path_power) {
428   int returnValue = 0;
429 #ifdef ASSISTED
```

```
430     int existingPath;
431     double cost = (double)sink_dist + log((sensor_power/path_power));
432 
433     int sinkLocation = check_for_sink(s, sink_gid);
434     if (sinkLocation != -1) {
435       existingPath = check_path_update(s, sinkLocation, sink_path);
436       if (existingPath != -1) {
437         s->sink_cost[sinkLocation][existingPath] = cost;
438         returnValue = 1;
439       }
440     }
441 
442     // Ensure that the list is sorted. In practice power is only reduced.
443     // Therefore it can only lose priority.
444     // Update a single sink first.
445     unsigned int i;
446     for (i = existingPath; i < MAX_PATHS_PER_SINK-1; ++i) {
447       if (s->sink_cost[sinkLocation][i] > s->sink_cost[sinkLocation][i+1]) {
448         // swap paths.
449         swap_paths(s, sinkLocation, i, i+1);
450       } else {
451         break;
452       }
453     }
454 
455     // Now update the sinks.
456     for (i = sinkLocation; i < MAX_SINKS-1; ++i) {
457       if (s->sink_cost[i][0] > s->sink_cost[i+1][0]) {
458         swap_sinks(s, i, i+1);
459       }
460     }
461 
462   #endif
463 
464     return returnValue;
465   }
466   /**
467    * Removes a sink from the list of available sinks
468    *
469    * @param s The sensor to remove the sink from.
470    * @param sink_gid The gid of the sink to remove.
471    *
472    * @return 0 if the sink was not found. 1 if found and removed.
473    */
474   int remove_sink(sensor_state *s, double sink_gid) {
475     int returnValue = 0;
476     int sinkLocation = check_for_sink(s, sink_gid);
477     if (sinkLocation != -1) {
478       returnValue = 1;
479 
480       // Set all values for that sink to unknown
481       s->sink_gid[sinkLocation] = UNKNOWN;
```

```
482       int i = 0;
483       for (i = 0; i < MAX_PATHS_PER_SINK; ++i) {
484         s->sink_path[sinkLocation][i] = UNKNOWN;
485         s->sink_dist[sinkLocation][i] = UNKNOWN;
486       }
487
488       // now shift all other sinks to condense array
489       for (i = sinkLocation; i < MAX_SINKS-1; ++i) {
490         if (s->sink_gid[i+1] != UNKNOWN) {
491           swap_sinks(s, i, i+1);
492         }
493       }
494     }
495     return returnValue;
496   }
497   #endif
498
499   #ifndef DIJKSTRA
500   #ifndef COBWEB
501
502   /**
503    * compares two messages to determine if they are the same.
504    *
505    * @param ms1 A message to compare
506    * @param ms2 Another message to compare.
507    *
508    * @return 0 if the messages are not the same. 1 if they are the same.
509    */
510   int msg_equal(sensor_message *ms1, sensor_message *ms2) {
511     if (ms1 != NULL && ms2 != NULL &&
512         ms1->source_gid == ms2->source_gid && ms1->nonce == ms2->nonce) {
513       return 1;
514     } else {
515       return 0;
516     }
517   }
518
519   /**
520    * Determines whether the sensor given has not received a response
521    * canceling a wait on a message previously sent out.
522    *
523    * @param s The sensor to check.
524    * @param msg The message to check if we are still waiting on it.
525    *
526    * @return The index of the message in the array if still waiting.
527    *         -1 if the sensor is not waiting on a response.
528    */
529   int stillWaiting(sensor_state *s, sensor_message *msg) {
530     int returnValue = -1;
531
532     int i = 0;
533     for (i = 0; i < MAX_WAIT; ++i) {
```

```
534       if (msg_equal(msg, s->wait_events[i]) == 1) {
535         returnValue = i;
536         break;
537       }
538     }
539     return returnValue;
540 }
541 /**
542  * Adds an event to the list of wait events on the sensor.
543  *
544  * @param s the sensor that will keep track of the event
545  * @param e the event to keep track of.
546  * @param lp The logical processor associated with the sensor.
547  *           Used for debugging purposes.
548  */
549 void save_event(sensor_state *s, tw_event *e, tw_lp *lp) {
550     if (s->num_wait_events == MAX_WAIT) {
551      printf("%li ERROR: MAX_WAIT_SIZE ACHIEVED\n", lp->gid);
552     } else {
553       sensor_message *m = tw_event_data(e);
554       s->wait_events[s->num_wait_events] = m;
555       s->num_wait_events++;
556     }
557 #ifdef VERBOSE
558     printf("%li savewaitct: %d\n", lp->gid, s->num_wait_events);
559 #endif
560 }
561
562 /**
563  * removes an event with the given source_gid/nonce combo from the sensor's
564  * wait list.
565  *
566  * @param s The sensor that was keeping track of the event.
567  * @param source_gid The source gid associated with the data.
568  * @param nonce the nonce associated with the data.
569  * @param lp The logical processor associated with the sensor.
570  *           Used for debugging purposes.
571  *
572  * @return A value of 1 if an event was removed, 0 if it was not removed.
573  */
574 int remove_event(sensor_state *s, double source_gid, double nonce, tw_lp *lp) {
575     int returnValue = 0;
576     int i = 0;
577     for (i = 0; i < s->num_wait_events; ++i) {
578       sensor_message *msg = s->wait_events[i];
579       if (msg->source_gid == source_gid &&
580           msg->nonce == nonce) {
581         returnValue = 1;
582         // remove the message from tracking by compacting the array
583         for (; i < s->num_wait_events; ++i) {
584           s->wait_events[i] = s->wait_events[i+1];
585         }
```

```
586         s->num_wait_events -= 1;
587         break;
588       }
589     }
590 #ifdef VERBOSE
591   printf("%li remwaitct: %d\n", lp->gid, s->num_wait_events);
592 #endif
593   return returnValue;
594 }
595
596 #endif
597 #endif
598
599 /**
600  * Prints infomation about messages in a standard format
601  *
602  * @param sourceGid The source of the message
603  * @param destGid The destination of the message
604  * @param time The time the message is sent
605  */
606 void printMessageInfo(long sourceGid, long destGid, double time) {
607   printf("EVENT:GID:%li:message:%li:%G\n", sourceGid, destGid, time);
608 }
609
610
611 /**
612  * Prints information about a sensor's state (awake, asleep)
613  *
614  * @param gid The id of the sensor
615  * @param time The time the state changed
616  * @param type The type of information being printed
617  * @param extra Any Extra information
618  * @param status The status of the sensor
619  */
620 void printStateInfo(long gid, double time, char *type, int extra,
621     int status) {
622   printf("STATE:GID:%li:%G:%s:%d:%d\n", gid, time, type, extra, status);
623 }
624
625 #endif
```

Listing A.3: Source Code

```
 1 /**
 2  * @file definitions.h
 3  * @brief Defintions that are common to all simulations.
 4  * @author Thomas Reale
 5  * @date 12-2011
 6  * @version 1.0
 7  *
 8  *
 9  * Defines variables and pre processor variables for use in the program.
10  */
```

```
11
12 #ifndef WSN_DEFINITIONS
13 #define WSN_DEFINITIONS
14
15 #include<ross.h>
16
17
18
19
20 /** The total number of sensors */
21 static int gnum_sensors = 99;
22
23 /** The number of logical processors per processing engine */
24 static tw_lpid  nlp_per_pe;
25
26 /** @def DEST_ALL
27  * Used to define a message being sent to all nearby */
28 #define DEST_ALL -2
29
30 /** @def UNKNOWN
31  * Used to identify no known sensor */
32 #define UNKNOWN -1
33
34 /** @def TRUE
35  * Used to indicate a true value */
36 #define TRUE 1
37
38 /** @def FALSE
39  * Used to indicate a false value */
40 #define FALSE 0
41
42 /**
43  * @\defgroup WSN_Physical_System_Variables
44  * Defines variables that pertain to the area the sensors are distributed in.
45  * @{
46  */
47 /** The area to distribute the sensors */
48 static int        grid_width = 10;
49 /** @} */
50
51 /**
52  * @\defgroup WSN_Sensor_Attributes
53  * Physical attributes common to all sensors
54  * @{
55  */
56 /** The range of each sensor at full power */
57 static int        sensor_range = 1;
58 /** The starting power of each sensor */
59 static int        sensor_power = 10;
60 /** @} */
61
62
```

```
63  /**
64   * @\defgroup WSN_Timing_Attributes
65   * Defines the average time that events take in a sensor
66   * @{
67   */
68  /** The time to spend sleeping */
69  static int       gsleep_time = 5;
70  /** The time to spend awake */
71  static int       gawake_time = 110;
72
73  /** A temp variable used to allow the delay to be selected via comand line */
74  static int transmit_delay_tenths = 1;
75  /** The delay it takes to send a message to neighbors with negligible size */
76  static double    transmission_delay = 5;
77  /** The standard deviation for a normal distribution when transmitting data */
78  static double    std = 0.1;
79  /** @} */
80
81  /**
82   * @\defgroup WSN_Data_Attributes
83   * Attributes of the data that is collected by the sensor
84   * @{
85   */
86  /** The amount of data generated (bits) at a time */
87  static int       data_size = 250000;
88  /** The time between data being generated */
89  static int       gdata_gen_rate = 100;
90  /** The speed (bits/sec) at which sensors can transmit data */
91  static int       gtransmit_rate = 250000;
92  /** The maximum amount of bits a sensor cans store */
93  static int       sensor_storage = 1048576;
94  /** @} */
95
96  /**
97   * @\defgroup WSN_Sensor_properties
98   * physical attributes
99   * @{
100  */
101 /** The amount of power used in transmission in uA*/
102 static int       transmit_power = 14000;
103 /** The amount of power used in active mode in uA */
104 static int       active_power = 8000;
105 /** The amount of power used in sleep mode in uA */
106 static int       sleep_power = 20;
107 /** @} */
108 /**
109  * @\defgroup WSN_Statistics
110  * Statistics collected by the simulation to be displayed upon completion
111  * @{
112  */
113 /** Average time awake for a single sensor */
114 static tw_stime awake_time_avg = 0;
```

```
115 /** Average time asleep for a single sensor */
116 static tw_stime asleep_time_avg = 0;
117 /** Average transmit time for a single sensor */
118 static tw_stime transmit_time_avg = 0;
119 /** Maximum transmit time for a single sensor */
120 static tw_stime transmit_time_max = 0;
121
122 /** The sum of the times it takes data to get from the originator
123  * to the sink */
124 static long double transmission_path_time_total = 0;
125 /** number of times data is received by sink */
126 static long double transmission_recv_ct = 0;
127
128 /** average number of messages sent per sensor */
129 static double num_msg_avg = 0;
130 /** Maximum number of messages sent by a single sensor */
131 static double max_msg_single = 0;
132 /** Total number of messages sent */
133 static double total_msg_count = 0;
134
135 /** total data size sink received */
136 static double data_sink_received = 0;
137 /** Total data generated */
138 static double total_data_generated = 0;
139 /** Data that is lost due to overflow */
140 static double total_data_overflow = 0;
141
142 /** Power statistics */
143 /** The average power remaining over all sensors at end of simulation */
144 static double average_power_remaining = 0;
145 /** The number of sensors that have no power remaining at the end of the simulation*
146 static int number_dead_sensors = 0;
147
148 /** @def INIT_TS_TRANSMIT(mu)
149  * Defines the ts variable used for transmitting information
150  * using a gaussian distribution and adding a delay
151  */
152 #define INIT_TS_TRANSMIT(mu) unsigned int num_calls; ts = transmission_delay + tw_ra
153
154 /** @def INIT_TS_NORMAL(mu)
155  * Initializes the ts variable using a normal distribution with a standard
156  * deviation given.
157  */
158 #define INIT_TS_NORMAL(mu) unsigned int num_calls; ts =tw_rand_normal_sd(lp->rng, (n
159
160 #endif
```

Listing A.4: Source Code

```
1 INCLUDE_DIRECTORIES(${ROSS_SOURCE_DIR})
2
3 SET(wsn_srcs
4 wsn.c utils.h selfselect/messageUtils.h selfselect/selfselect.h selfselect/wsn.h dij
```

```
5
6 #Indicates dijkstra's_algorithm_should_be_used
7 #ADD_DEFINITIONS(-DDIJKSTRA)
8
9 #Variables that are used with self select
10 #ADD_DEFINITIONS(-DSELFSELECT)
11
12 #Indicates cobweb routing should be used
13 ADD_DEFINITIONS(-DCOBWEB)
14
15 #Indicates that assisted routing should be used
16 #ADD_DEFINITIONS(-DASSISTED)
17
18 #Indicates verbose mode
19 #ADD_DEFINITIONS(-DVERBOSE)
20
21 #Verbose for Awake/Asleep
22 #ADD_DEFINITIONS(-DVERBOSES)
23
24 ADD_EXECUTABLE(wsn ${wsn_srcs})
25
26 TARGET_LINK_LIBRARIES(wsn ROSS m)
```

A.3 Cobweb Files

Listing A.5: Source Code

```
1 /**
2  * @file cobweb/cobweb.h
3  * @brief Processes events for the Cobweb algorithm.
4  * @author Thomas Reale
5  * @date 12-2011
6  * @version 1.0
7  *
8  * Implements the Cobweb algorithm by processing all events that occur
9  * during the simulation.
10  */
11 #ifndef WSN_COBWEB_H
12 #define WSN_COBWEB_H
13
14 #include "wsn.h"
15 #include "../utils.h"
16 #include "messageUtils.h"
17
18 /** @def WHO_IS_SINK
19  * Who is the sink based on an lp's gid */
20 #define WHO_IS_SINK (lp->gid==1)
21 /** @def WHO_GEN_DATA
22  * Who generates data based on lp's gid */
23 #define WHO_GEN_DATA (lp->gid==0)
24
25 /**
26  * Initializes all variables in a single sensor.
```

```
27   *
28   * @param s The sensor to initialize.
29   */
30  void initializeVariables(sensor_state *s) {
31    s->total_awake_time = 0;
32    s->power = sensor_power;
33    s->state = AWAKE;
34    s->num_hops = UNKNOWN;
35    s->data_size = 0;
36    s->nearby_Sensors = NULL;
37    s->num_nearby = 0;
38    s->last_state_change = 0;
39  }
40
41  /**
42   * The events that will occur when the INIT event is trigered.
43   * This event is called once at the very beginning of the simulation.
44   *
45   * Sets up which sensors identify as sinks, which sensors generate data
46   * as well as initiating awake/sleep cycle. (currently doesn't actually wake/sleep 
47   *
48   * @param s The sensor associated with this event
49   * @param bf The time warp bit field.
50   * @param msg The message associated with this event.
51   * @param lp The logical processor with the sensor
52   */
53  void sensor_init(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
54    findNearbySensors(s, lp->gid);
55
56    if (WHO_IS_SINK) {
57      sendMessage(s, lp, SINK_DISCOVERED);
58    }
59
60    if (WHO_GEN_DATA) {
61      // start collecting data
62      sendSelfMessage(s, lp, DATA_GEN, NULL);
63    }
64
65    // All but the sink sleep
66    if (!(WHO_IS_SINK)) {
67      // Start the Awake Sleep Cycle
68      s->last_state_change = 0;
69      sendSelfMessage(s, lp, SLEEP, NULL);
70    }
71  }
72
73  /**
74   * Processes an awake event on a sensor.
75   * The sensor enters the awake state and schedules a sleep event.
76   *
77   * @param s The sensor associated with the event
78   * @param bf The time warp bit field for reverse computation.
```

```
79   * @param msg The sensor message for the event.
80   * @param lp The logical processor with this sensor
81   */
82  void awake_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
83                   tw_lp *lp) {
84
85    // Update Stats
86    double timeSpent = tw_now(lp) - s->last_state_change;
87    s->total_asleep_time += timeSpent;
88    s->times_asleep++;
89
90    // Change state to AWAKE
91    s->state = AWAKE;
92    s->last_state_change = tw_now(lp);
93    s->power -= power_loss(timeSpent, sleep_power);
94
95    // Create an event to go to sleep
96    sendSelfMessage(s, lp, SLEEP, NULL);
97
98  }
99
100 /**
101  * Processes a sleep event on a sensor.
102  * Indicates that the sensor should go to sleep if it is not processing.
103  * If it is, starts another awake cycle.
104  *
105  * @param s The sensor associated with the event
106  * @param bf The time warp bit field for reverse computation.
107  * @param msg The sensor message for the event.
108  * @param lp The logical processor with this sensor
109  */
110 void sleep_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
111                  tw_lp *lp) {
112   s->state = SLEEP;
113
114   // Update Stats
115   double timeSpent = tw_now(lp) - s->last_state_change;
116   s->total_awake_time += timeSpent;
117   s->times_awake++;
118   s->power -= power_loss(timeSpent, active_power);
119
120   // Create an event to wake up
121   s->last_state_change = tw_now(lp);
122   sendSelfMessage(s, lp, AWAKE, NULL);
123 }
124
125 /**
126  * Processes a sink loc event on a sensor
127  *
128  *
129  * For RC, if c0 is -1, no change made.
130  * c0 = index changed,
```

```
131    * c1 = old num_hops sensor
132    * msg->saved_hops = old num_hops nearby
133    * msg->saved_power = old power nearby
134    *
135    * @param s The sensor associated with the event
136    * @param bf The TW bit field for RC.
137    * @param msg The message received
138    * @param lp The logical processor with this sensor
139    */
140   void sink_loc_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
141     // Indicates a sink is broadcasting it's position
142     bf->c0 = -1;
143
144     // update the neighbor information. A neighbor only broadcasts
145     // it's location if it is a bettor dist to the sink.
146     int i = 0;
147     for (i = 0; i < s->num_nearby; ++i) {
148       if (s->nearby_Sensors[i].gid == msg->last_gid) {
149
150         // Store info for RC
151         bf->c0 = i;
152         bf->c1 = s->num_hops;
153         msg->saved_hops = s->nearby_Sensors[i].hopCount;
154         msg->saved_power = s->nearby_Sensors[i].resEnergy;
155
156         // Update the sensor
157         s->nearby_Sensors[i].hopCount = msg->num_hops-1;
158         s->nearby_Sensors[i].resEnergy = msg->last_power;
159
160         break;
161       }
162     }
163
164     // Only update information if it is a better route.
165     if (s->num_hops == UNKNOWN || s->num_hops > msg->num_hops) {
166       // Store data for rc. c0 = index, c1 = old num_hops
167
168       s->num_hops = msg->num_hops;
169
170       // Broadcast the message to all surrounding sensors
171       sendMessage(s, lp, SINK_LOC);
172     }
173
174     // start sending data (if any) to the sink
175     send_data(s, bf, lp, NULL);
176   }
177
178   /**
179    * Processes a sink lost event on a sensor
180    * Removes the sink from the sensor and broadcasts SINK_LOST to neighbors.
181    *
182    * @param s The sensor associated with the event
```

```
183  * @param bf The time warp bit field for reverse computation.
184  * @param msg The sensor message for the event.
185  * @param lp The logical processor with this sensor
186  */
187 void sink_lost_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
188   // Indicates that a sink has been lost
189   if (s->num_hops != UNKNOWN) {
190     s->num_hops = UNKNOWN;
191     if (s->state == TRANSMIT) {
192       s->state = AWAKE;
193     }
194     sendMessage(s, lp, SINK_LOST);
195   }
196 }
197
198 /**
199  * Processes a data gen event on a sensor
200  * Adds data to the sensors storage and starts sending data.
201  * If this sensor is the sink just updates statistics indicating
202  * completed reception. Schedules a DATA_GEN in the future.
203  *
204  * @param s The sensor associated with the event
205  * @param bf The time warp bit field for reverse computation.
206  * @param msg The sensor message for the event.
207  * @param lp The logical processor with this sensor
208  */
209 void data_gen_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
210   // Indicates that data has been generated
211   s->data_size += data_size;
212   // generate data in the future
213   sendSelfMessage(s, lp, DATA_GEN, NULL);
214   printStateInfo(lp->gid, tw_now(lp), "DATA", s->data_size, s->state);
215   send_data(s, bf, lp, NULL);
216 }
217
218 /**
219  * Processes a data received event on a sensor
220  * If this sensor has received data forward it on.
221  *
222  * @param s The sensor associated with the event
223  * @param bf The time warp bit field for reverse computation.
224  * @param msg The sensor message for the event.
225  * @param lp The logical processor with this sensor
226  */
227 void data_received_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp)
228 #ifdef VERBOSE
229   printf("%li data_received\n", lp->gid);
230 #endif
231
232   s->data_size += msg->data_size;
233   send_data(s, bf, lp, msg);
234 }
```

```
235
236  /**
237   * Indicates that a transmission has completed. Wakes up the sensor for
238   * further communication.
239   *
240   * @param s The sensor that completed transmission.
241   * @param bf The time warp bit field for reverse computation.
242   * @param msg The sensor message for the event.
243   * @param lp The logical processor associated with tihs sensor.
244   */
245  void transmit_complete_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp
246  #ifdef VERBOSE
247    printf("%li transmit complete\n", lp->gid);
248  #endif
249    s->state = AWAKE;
250  }
251
252
253
254
255
256
257
258
259
260
261
262
263
264  /**
265   * Reverse compiler event for initializing a sensor
266   *
267   * @param s The sensor associated with the event.
268   * @param bf The time warp bit field for reverse computation.
269   * @param msg The sensor message for the event.
270   * @param lp The logical processor with this sensor.
271   */
272  void sensor_init_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
273                      tw_lp *lp) {
274    // Should never be called. Even if it is this function only resets
275    // data
276  }
277
278  /**
279   * Reverse compiler event for a sensor going to sleep
280   *
281   * @param s The sensor associated with the event.
282   * @param bf The time warp bit field for reverse computation.
283   * @param msg The sensor message for the event.
284   * @param lp The logical processor with this sensor.
285   */
286  void sleep_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
```

```
287                       tw_lp *lp) {
288 }
289
290 /**
291  * Reverse compiler event for a sensor waking up
292  *
293  * @param s The sensor associated with the event.
294  * @param bf The time warp bit field for reverse computation.
295  * @param msg The sensor message for the event.
296  * @param lp The logical processor with this sensor.
297  */
298 void awake_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
299                       tw_lp *lp) {
300 }
301
302 /**
303  * Reverse compiler event for a sink located
304  *
305  * @param s The sensor associated with the event.
306  * @param bf The time warp bit field for reverse computation.
307  * @param msg The sensor message for the event.
308  * @param lp The logical processor with this sensor.
309  */
310 void sink_loc_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
311                          tw_lp *lp) {
312   // If any changes were made
313   if (bf->c0 != -1) {
314     s->nearby_Sensors[bf->c0].hopCount = msg->saved_hops;
315     s->nearby_Sensors[bf->c0].resEnergy = msg->saved_power;
316
317     s->num_hops = bf->c1;
318   }
319 }
320 /**
321  * Reverse compiler event for a sink lost
322  *
323  * @param s The sensor associated with the event.
324  * @param bf The time warp bit field for reverse computation.
325  * @param msg The sensor message for the event.
326  * @param lp The logical processor with this sensor.
327  */
328 void sink_lost_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
329                           tw_lp *lp) {
330
331 }
332
333 /**
334  * Reverse compiler event for data generation
335  *
336  * @param s The sensor associated with the event.
337  * @param bf The time warp bit field for reverse computation.
338  * @param msg The sensor message for the event.
```

```
339  * @param lp The logical processor with this sensor.
340  */
341 void data_gen_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
342                          tw_lp *lp) {
343   // Check to see if data was sent
344   if (WHO_IS_SINK) {
345     // If this is the sink, update stats.
346     data_sink_received -= bf->c11;
347     transmission_path_time_total -= tw_now(lp) - msg->start_time;
348     transmission_recv_ct--;
349   }
350   if (bf->c10 == 1) {
351     tw_rand_reverse_unif(lp->rng);
352     s->messages_sent--;
353   } else if (!WHO_IS_SINK) {
354     // remove the data
355     s->data_size -= data_size;
356   }
357 }
358 /**
359  * Reverse compiler event for data being received
360  *
361  * @param s The sensor associated with the event.
362  * @param bf The time warp bit field for reverse computation.
363  * @param msg The sensor message for the event.
364  * @param lp The logical processor with this sensor.
365  */
366 void data_received_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
367                               tw_lp *lp) {
368   if (WHO_IS_SINK) {
369     // If this is the sink, update stats.
370     data_sink_received -= msg->data_size;
371     transmission_path_time_total -= tw_now(lp) - msg->start_time;
372     transmission_recv_ct--;
373   }
374   // Check to see if data was forwarded on.
375   if (bf->c10 == 1) {
376     s->state = AWAKE;
377     tw_rand_reverse_unif(lp->rng);
378     s->messages_sent--;
379   } else {
380     // remove data as we no longer received it
381     s->data_size -= data_size;
382   }
383 }
384 /**
385  * Reverse compiler event for data being received
386  *
387  * @param s The sensor associated with the event.
388  * @param bf The time warp bit field for reverse computation.
389  * @param msg The sensor message for the event.
390  * @param lp The logical processor with this sensor.
```

```
391    */
392  void transmit_complete_event_rc(sensor_state *s, tw_bf *bf,
393                                  sensor_message *msg, tw_lp *lp) {
394    // The status is back to transmitting
395    s->state = TRANSMIT;
396  }
397  #endif
```

Listing A.6: Source Code

```
1   /**
2    * @file cobweb/messageUtils.h
3    * @brief Messaging functions for use with the Cobweb algorithm.
4    * @author Thomas Reale
5    * @date 12-2011
6    * @version 1.0
7    *
8    * Functions to generate and send events to other sensors.
9    *
10   */
11  #ifndef WSN_MESSAGE_UTILS_H
12  #define WSN_MESSAGE_UTILS_H
13
14  #include "../utils.h"
15
16  /** Generates an event indicating transmission is complete and the sensor
17   * can change to the AWAKE state.
18   *
19   * @param lp The logical processor associated with the sensor
20   * @param transmit_time How long the sensor is transmitting
21   *
22   * @return An event indicating the sensor should wake up.
23   */
24  tw_event *genTransmitCompleteMessage(tw_lp *lp, tw_stime transmit_time) {
25    sensor_message *m;
26    tw_event *e;
27
28    e = tw_event_new(lp->gid, transmit_time, lp);
29    m = tw_event_data(e);
30    m->type = TRANSMIT_COMPLETE;
31
32    return e;
33  }
34
35  /**
36   * Generates a Data Gen message
37   *
38   * @param lp The logical processor
39   *
40   * @return The created event.
41   */
42  tw_event *genDataGenMessage(tw_lp *lp) {
43    tw_stime ts = gdata_gen_rate;
```

```
44   tw_event *e;
45   sensor_message *m;
46
47   e = tw_event_new(lp->gid, ts, lp);
48   m = tw_event_data(e);
49   m->type = DATA_GEN;
50
51   return e;
52 }
53
54 /**
55  * Generates a sleep message to go to sleep in the future
56  *
57  * @param lp The logical processor
58  *
59  * @return The created event
60  */
61 tw_event *genSleepMessage(tw_lp *lp) {
62   return NULL;
63 }
64 /**
65  * Generates an awake message to wake up in the future
66  *
67  * @param lp The logical processor
68  *
69  * @return The created event
70  */
71 tw_event *genAwakeMessage(tw_lp *lp) {
72   return NULL;
73 }
74
75 /**
76  * Sends a message to self only.
77  *
78  * @param s The state of the sensor.
79  * @param lp The logical processor associated with that sensor
80  * @param type The type of the message
81  * @param m A sensor message that triggered this event. Currently included
82  *              to be consisten with other simulations.
83  */
84 void sendSelfMessage(sensor_state *s, tw_lp *lp, sensor_event_t type,
85                      sensor_message *m) {
86   tw_event *e = NULL;
87
88   switch(type) {
89     case SLEEP:
90       e = genSleepMessage(lp);
91       break;
92     case AWAKE:
93       e = genAwakeMessage(lp);
94       break;
95     case DATA_GEN:
```

```
 96         e = genDataGenMessage(lp);
 97         break;
 98       default:
 99         break;
100     }
101 
102     if (e != NULL) {
103       tw_event_send(e);
104     }
105 
106 }
107 
108 /**
109  * Generates a sinc_loc event
110  *
111  * @param s The sensor state containing information to put in message.
112  * @param dest The destination gid.
113  * @param offset_time How long it will take this event to complete.
114  * @param lp This logical processor.
115  *
116  * @return The created sinc_loc event
117  */
118 tw_event *genSinkLocMessage(sensor_state *s, double dest, tw_stime offset_time,
119                             tw_lp *lp) {
120   tw_event *e;
121   sensor_message *m;
122 
123   e = tw_event_new(dest, offset_time, lp);
124   m = tw_event_data(e);
125   m->type = SINK_LOC;
126   m->num_hops = s->num_hops+1;
127   m->last_power = s->power;
128   m->last_gid = lp->gid;
129   m->dest = DEST_ALL;
130   return e;
131 }
132 
133 
134 /**
135  * Generates a data received message at the given dest node.
136  * Bases the time it takes to transmit off of the transmit_rate variable
137  * and the amount of data.
138  *
139  *
140  * @param gid The GID of the destination node.
141  * @param lp The logical processor initiating the request.
142  * @param msg A message containing any information about the new messag to be genera
143  * @param s A sensor state used for statistics purposes
144  *
145  * @return the created event.
146  */
147 tw_event *genDataReceivedMessage(double gid, tw_lp *lp, sensor_message *msg,
```

```
148                                       sensor_state *s) {
149   tw_stime ts;
150   INIT_TS_TRANSMIT((msg->data_size / gtransmit_rate));
151   s->total_transmit_time += ts;
152
153   tw_event *transmitevent = genTransmitCompleteMessage(lp, ts);
154   tw_event_send(transmitevent);
155
156   sensor_message *m;
157   tw_event *e;
158
159   e = tw_event_new(gid, ts, lp);
160   m = tw_event_data(e);
161   m->type = DATA_RECEIVED;
162   m->data_size = msg->data_size;
163   m->start_time = msg->start_time;
164   m->dest = gid;
165
166   // send self an event to indicate transmission is complete
167   sendSelfMessage(s, lp, TRANSMIT_COMPLETE, NULL);
168   return e;
169
170 }
171
172 /**
173  * Generates a sink lost event
174  *
175  * @param dest The destination gid
176  * @param lp This logical processor
177  *
178  * @return The created event
179  */
180 tw_event *genSinkLostMessage(double dest, tw_lp *lp) {
181   return NULL;
182 }
183
184 /**
185  * Sends a message originating from s to all nearby sensors.
186  *
187  * @param s The originating sensor
188  * @param lp the associated lp.
189  * @param type The event type to send
190  */
191 void sendMessage(sensor_state *s, tw_lp *lp, sensor_event_t type) {
192   //sensor_message *m;
193   tw_event *e = NULL;
194
195   // Take care of events that alter the state of the originating sensor
196   switch (type) {
197     case SINK_DISCOVERED:
198       s->num_hops = 0;
199       type = SINK_LOC;
```

```
200         break;
201       default:
202         break;
203     }
204 
205     // Send the requested message if valid type to all nearby neighbors
206     int i = 0;
207     tw_stime ts;
208     INIT_TS_TRANSMIT(transmission_delay);
209     if (ts < 0) {
210       ts *= -1;
211     }
212 
213     s->total_transmit_time += ts;
214     s->power -= power_loss(ts, transmit_power);
215 
216     for (i = 0; i < s->num_nearby; ++i) {
217       switch (type) {
218         case SINK_LOC:
219           e = genSinkLocMessage(s, s->nearby_Sensors[i].gid, ts, lp);
220           break;
221         case SINK_LOST:
222           e = genSinkLostMessage(s->nearby_Sensors[i].gid, lp);
223           break;
224         default:
225           // These types should not be associated with this type
226           // of message sending.
227           break;
228       }
229       if (e != NULL) {
230         tw_event_send(e);
231         printMessageInfo(lp->gid, s->nearby_Sensors[i].gid, tw_now(lp));
232         // The multiple messages are used for the simulation. In reality
233         // this would be done by a single transmission.
234         s->messages_sent++;
235       } else {
236         printf("WARNING: Event Not Sent\n");
237       }
238     }
239 }
240 
241 /**
242  * Finds the best neighbor to send data to based on hops and energy.
243  *
244  * @param s The sensor to find the neighbor for.
245  *
246  * @return The best neighbor.
247  */
248 long findBestNeighbor(sensor_state *s) {
249     // find the sensor to send the data to
250     int minHop = s->nearby_Sensors[0].hopCount;
251     double minEnergy = s->nearby_Sensors[0].resEnergy;
```

```
252     long minGid = s->nearby_Sensors[0].gid;
253
254     int i = 0;
255     for (i = 1; i < s->num_nearby; ++i) {
256       if (s->nearby_Sensors[i].hopCount < minHop) {
257         minHop = s->nearby_Sensors[i].hopCount;
258         minEnergy = s->nearby_Sensors[i].resEnergy;
259         minGid = s->nearby_Sensors[i].gid;
260       } else if (s->nearby_Sensors[i].hopCount == minHop) {
261         if (s->nearby_Sensors[i].resEnergy < minEnergy) {
262           minHop = s->nearby_Sensors[i].hopCount;
263           minEnergy = s->nearby_Sensors[i].resEnergy;
264           minGid = s->nearby_Sensors[i].gid;
265         }
266       }
267     }
268
269     return minGid;
270   }
271
272   /**
273    * Attempts to send data to a known sink.
274    * If no sink is known, broadcasts a SINK_LOST message to update all around.
275    *
276    * FOR RC:
277    * if (bf->c10 == 1) started transmitting
278    * bf->c11 is data size for sink.
279    *
280    * @param s The sensor state.
281    * @param bf The time warp bit field.
282    * @param lp The processor associated with the lp
283    * @param msg If not null, indicates there is information conatined within
284    *            containing statistics information to forward.
285    */
286   void send_data(sensor_state *s, tw_bf *bf, tw_lp *lp, sensor_message *msg) {
287     bf->c10 = 0;
288     // If there is data, and sink distance known send data
289   #ifdef VERBOSE
290     printf("%li send: state:%d:numhops:%d:data_size:%G\n", lp->gid, s->state,
291         s->num_hops, s->data_size);
292   #endif
293     if (s->state != TRANSMIT && s->num_hops >= 0 && s->data_size > 0) {
294
295       s->state = TRANSMIT;
296       // Indicate that data is being sent for RC.
297       bf->c10 = 1;
298
299       int sentdata = s->data_size;
300
301       if (msg != NULL) {
302         sentdata = msg->data_size;
303       }
```

```
304      s->data_size -= sentdata;
305      bf->c11 = sentdata;
306
307      // Only forward if this is not connected to sink
308      if (s->num_hops > 0) {
309        if (s->num_nearby > 0) {
310
311          long neighbor = findBestNeighbor(s);
312
313          int isNew = 0;
314          if (msg == NULL) {
315            isNew = 1;
316            msg = (sensor_message *)malloc(sizeof(sensor_message));
317            msg->data_size = sentdata;
318            msg->start_time = tw_now(lp);
319          }
320
321          tw_event *event = genDataReceivedMessage(neighbor, lp, msg, s);
322          tw_event_send(event);
323          printMessageInfo(lp->gid, neighbor, tw_now(lp));
324          s->messages_sent++;
325
326          // free memory if needed.
327          if (isNew == 1) {
328            free(msg);
329            msg = NULL;
330          }
331        }
332      } else {
333        printf("SINK_GOT_DATA: size:%d\n", sentdata);
334        // this is the sink. update the stat
335        data_sink_received += sentdata;
336        tw_stime start_time;
337        if (msg == NULL) {
338          start_time = tw_now(lp);
339        } else {
340          start_time = msg->start_time;
341        }
342        transmission_path_time_total += tw_now(lp) - start_time;
343        transmission_recv_ct ++;
344        s->state = AWAKE;
345      }
346    } else if (s->data_size == 0) {
347        s->state = AWAKE;
348    }
349  }
350
351
352  #endif
```

Listing A.7: Source Code

```
1  /**
```

```
2   * @file cobweb/wsn.h
3   * @brief Defines the structures and any Cobweb specific variables.
4   * @author Thomas Reale
5   * @date 12-2011
6   * @version 1.0
7   *
8   * Defines the message structure, the sensor structure, and possible events.
9   *
10  */
11 #ifndef INC_wsn_h
12 #define INC_wsn_h
13
14 #include <ross.h>
15
16 /** typedef to identify an event */
17 typedef enum sensor_event_t sensor_event_t;
18 /** typedef to define a sensor's state */
19 typedef struct sensor_state sensor_state;
20 /** typedef to define a message between sensors */
21 typedef struct sensor_message sensor_message;
22 /** typedef to define a location */
23 typedef struct Location Location;
24 /** typedef to define a sensor structure */
25 typedef struct Sensor Sensor;
26 /** typedef to define a neighbor structure */
27 typedef struct Neighbor Neighbor;
28
29 /** Keeps track of all sensors to allow to simulate distance between sensors */
30 static Sensor **sensors_list;
31
32
33 /** The possible events associated with a sensor */
34 enum sensor_event_t
35 {
36   SLEEP = 1,
37   AWAKE,
38   DEAD,
39   TRANSMIT,
40   TRANSMIT_COMPLETE,
41   SINK_LOC,
42   SINK_LOST,
43   SINK_DISCOVERED,
44   DATA_GEN,
45   DATA_RECEIVED,
46   INIT,
47 };
48
49 /** contains a cartesian coordinate */
50 struct Location {
51   /** The horizontal position of a sensor */
52   long x;
53   /** The vertical position of a sensor */
```

```
54    long y;
55  };
56
57  /** Information about a sensor for others to use for communication */
58  struct Sensor {
59    /** The location of this sensor */
60    Location location;
61    /** The global ID of this sensor */
62    long gid;
63  };
64
65  /** Information on neighbor nodes */
66  struct Neighbor {
67    /** The global ID of the neighbor */
68    long gid;
69    /** The energy remaining in the sensor */
70    double resEnergy;
71    /** The hop count to the sink */
72    int hopCount;
73  };
74
75  /**
76   * Contains the information related to a single sensor
77   * including statistics.
78   */
79  struct sensor_state
80  {
81    /** Power remaining in the sensor */
82    double power;
83    /** The state of the sensor (AWAKE/SLEEP/TRANSMIT) */
84    int state;
85    /** Location of the sensor */
86    Location location;
87    /** The number of hops to the nearest sink */
88    int num_hops;
89    /** The data size on the device in bytes */
90    double data_size;
91    /** Information about the neighboring sensors */
92    Neighbor *nearby_Sensors;
93    /** The number of nearby sensors */
94    int num_nearby;
95    /** Last time state changes used for updating purposes */
96    tw_stime last_state_change;
97
98    // Used for statistics
99    /** Total number of times awake */
100   double times_awake;
101   /** Total number of times asleep */
102   double times_asleep;
103   /** Total number of messages sent */
104   double messages_sent;
105   /** Total awake time */
```

```
106     tw_stime total_awake_time;
107     /** Total asleep time */
108     tw_stime total_asleep_time;
109     /** Total transmit time */
110     tw_stime total_transmit_time;
111 };
112
113 /**
114  * A message to trigger events
115  */
116 struct sensor_message
117 {
118     /** The type of event this message is associated with */
119     sensor_event_t type;
120     /** The data size contained in this message (DATA RECEIVED/DATA GEN) */
121     double data_size;
122     /** The destination gid of this message or DEST_ALL */
123     double dest;
124
125     /** Number of hops to sink */
126     int num_hops;
127     /** gid of sensor that sent this message. */
128     double last_gid;
129     /** power of the last sensor this message is from */
130     double last_power;
131     /** Time initial message sent */
132     tw_stime start_time;
133
134     // RC fields
135     /** Any GID needing saving */
136     double saved_gid;
137     /** Any power needing saving */
138     double saved_power;
139     /** Any num_hops needing saving */
140     int saved_hops;
141
142 };
143
144 #endif
```

A.4 Self Select Files

Listing A.8: Source Code

```
1 /**
2  * @file selfselect/selfselect.h
3  * @brief Processes events for the Self Select algorithm.
4  * @author Thomas Reale
5  * @date 12-2011
6  * @version 1.0
7  *
8  * Implements the Self Select algorithm by processing all events that occur
9  * during the simulation.
```

```
10  */
11 #ifndef WSN_SELFSELECT_H
12 #define WSN_SELFSELECT_H
13
14 #include "wsn.h"
15 #include "../utils.h"
16 #include "messageUtils.h"
17
18 /** @def WHO_IS_SINK
19  * Who is the sink based on an lp's gid */
20 #define WHO_IS_SINK (lp->gid==1)
21 /** @def WHO_GEN_DATA
22  * Who generates data based on lp's gid */
23 #define WHO_GEN_DATA (lp->gid==0)
24
25 /**
26  * Initializes all variables in a single sensor.
27  *
28  * @param s The sensor to initialize.
29  */
30 void initializeVariables(sensor_state *s) {
31   s->power = sensor_power;
32   s->nearby_Sensors = NULL;
33   s->num_nearby = 0;
34   s->num_wait_events = 0;
35   s->forwarding = NULL;
36   s->state = AWAKE;
37   s->trying_to_send = FALSE;
38   s->data_merging = FALSE;
39   s->duplicate_gid = -1;
40   s->last_state_change = 0;
41   s->wants_to_sleep = FALSE;
42
43   // Statistics
44   s->total_awake_time = 0;
45   s->total_asleep_time = 0;
46   s->total_transmit_time = 0;
47 }
48 /**
49  * The events that will occur when the INIT event is trigered.
50  * This event is called once at the very beginning of the simulation.
51  *
52  * Sets up which sensors identify as sinks, which sensors generate data
53  * as well as initiating awake/sleep cycle.
54  *
55  * @param s The sensor associated with this event
56  * @param bf The time warp bit field for reverse computation.
57  * @param msg The sensor message for the event.
58  * @param lp The logical processor with the sensor
59  */
60 void sensor_init(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
61   findNearbySensors(s, lp->gid);
```

```
62
63     if (WHO_IS_SINK) {
64       sendMessage(s, lp, SINK_DISCOVERED,NULL);
65     }
66
67     if (WHO_GEN_DATA) {
68       // start collecting data
69       sendSelfMessage(s, lp, DATA_GEN,NULL);
70     }
71
72     // All but the sink sleep
73     if (!(WHO_IS_SINK)) {
74       // Start the Awake Sleep Cycle
75       s->last_state_change = 0;
76       sendSelfMessage(s, lp, SLEEP, NULL);
77     }
78   }
79
80   /**
81    * Processes an awake event on a sensor.
82    * The sensor enters the awake state and schedules a sleep event.
83    *
84    * @param s The sensor associated with the event
85    * @param bf The time warp bit field for reverse computation.
86    * @param msg The sensor message for the event.
87    * @param lp The logical processor with this sensor
88    */
89   void awake_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
90                    tw_lp *lp) {
91
92     // Update Stats
93     double timeSpent = tw_now(lp) - s->last_state_change;
94     s->total_asleep_time += timeSpent;
95     s->times_asleep++;
96
97     // Change state to AWAKE
98     s->state = AWAKE;
99     s->last_state_change = tw_now(lp);
100    s->power -= power_loss(timeSpent, sleep_power);
101
102    // Create an event to go to sleep
103    sendSelfMessage(s, lp, SLEEP, NULL);
104
105  }
106
107  /**
108   * Sets the sensor to a sleep state and schedules an AWAKE event in the future.
109   *
110   * @param s The sensor going to sleep.
111   * @param bf The TW bf for RC.
112   * @param lp The logical processor associated with this sensor.
113   */
```

```
114 void go_to_sleep(sensor_state *s, tw_bf *bf, tw_lp *lp) {
115
116   s->state = SLEEP;
117   s->wants_to_sleep = FALSE;
118
119   // Update Stats
120   double timeSpent = tw_now(lp) - s->last_state_change;
121   s->total_awake_time += timeSpent;
122   s->times_awake++;
123   s->power -= power_loss(timeSpent, active_power);
124
125   // Create an event to wake up
126   s->last_state_change = tw_now(lp);
127   sendSelfMessage(s, lp, AWAKE, NULL);
128 }
129 /**
130  * Processes a sleep event on a sensor.
131  * If the sensor is not processing a message goes to sleep.
132  * Otherwise sets the wants_to_sleep variable to true in the sensor.
133  *
134  * @param s The sensor associated with the event
135  * @param bf The time warp bit field for reverse computation.
136  * @param msg The sensor message for the event.
137  * @param lp The logical processor with this sensor
138  */
139 void sleep_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
140                  tw_lp *lp) {
141   // Check to see if it is possible to go to sleep.
142   // Make sure not waiting to forward data
143   if (s->num_wait_events == 0 && s->forwarding == NULL) {
144     go_to_sleep(s, bf, lp);
145   } else {
146     s->wants_to_sleep = TRUE;
147   }
148 }
149
150 /**
151  * Processes a sink loc event on a sensor
152  * Stores the sink on the sensor and broadcasts SINK_LOC to neighbors.
153  *
154  * @param s The sensor associated with the event
155  * @param bf The time warp bit field for reverse computation.
156  * @param msg The sensor message for the event.
157  * @param lp The logical processor with this sensor
158  */
159 void sink_loc_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
160   // Indicates a sink is broadcasting it's position
161   // set the last hop gid to the path with the least
162   // distance to the sink
163   if (add_sink(s, msg->sink_gid, msg->last_gid, msg->num_hops, sensor_power)) {
164     // Broadcast the message to all surrounding sensors
165     sendMessage(s, lp, SINK_LOC,NULL);
```

```
166     }
167   }
168
169   /**
170    * Processes a sink lost event on a sensor.
171    * Removes the sink and broadcasts SINK_LOST to neighbors.
172    *
173    * @param s The sensor associated with the event
174    * @param bf The time warp bit field for reverse computation.
175    * @param msg The sensor message for the event.
176    * @param lp The logical processor with this sensor
177    */
178   void sink_lost_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
179     // Indicates that a sink has been lost
180     if (remove_sink(s, msg->sink_gid)) {
181       sendMessage(s, lp, SINK_LOST,NULL);
182     }
183   }
184
185   /**
186    * Processes a data gen event on a sensor
187    * Stores the data, attempts to send it to the sink if possible and creates an
188    * event to generate data in the future.
189    *
190    * @param s The sensor associated with the event
191    * @param bf The time warp bit field for reverse computation.
192    * @param msg The sensor message for the event.
193    * @param lp The logical processor with this sensor
194    */
195   void data_gen_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
196
197     if (!(WHO_IS_SINK)) {
198       // Indicates that data has been generated
199       s->data_size += data_size;
200       printStateInfo(lp->gid, tw_now(lp), "DATA", s->data_size, s->state);
201
202       send_data(s, lp, NULL);
203
204     } else {
205       // update stats
206       transmission_recv_ct++;
207       data_sink_received += data_size;
208     }
209
210     // generate data in the future
211     sendSelfMessage(s, lp, DATA_GEN, NULL);
212   }
213
214   /**
215    * Processes a data received event on a sensor
216    * If the sink has received this message, sends an ack to the previous sensor
217    * and updates statistics.
```

```
218   * If this sensor is not currently forwarding data and not waiting on this
219   * message start the backoff timer.
220   * If this sensor is not currently forwarding data but waiting on this message
221   * it is a duplicate and sends an ACK to the previous sensor to kill the path.
222   *
223   * @param s The sensor associated with the event
224   * @param bf The time warp bit field for reverse computation.
225   * @param msg The sensor message for the event.
226   * @param lp The logical processor with this sensor
227   */
228  void data_received_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
229                           tw_lp *lp) {
230    // IF this sensor is the sink, send an ACK to stop all others.
231    if (WHO_IS_SINK) {
232  #ifdef VERBOSE
233      printf("SINK_RECEIVED\n");
234  #endif
235      transmission_recv_ct++;
236      transmission_path_time_total += tw_now(lp) - msg->start_time;
237      data_sink_received += msg->data_size;
238
239      // Send an ack back to the previous owner.
240      tw_stime ts;
241      INIT_TS_TRANSMIT(transmission_delay);
242      tw_event *e = genAckMessage(msg->last_gid, ts, lp, msg);
243      tw_event_send(e);
244
245      printMessageInfo(lp->gid, msg->last_gid, tw_now(lp));
246
247    } else if (s->forwarding == NULL) {
248      // This is a new event. Store the data and start a backoff timer.
249      s->data_size += msg->data_size;
250      if (stillWaiting(s, msg) == -1) {
251        sendSelfMessage(s, lp, WAIT, msg);
252      } else {
253        // We have received a duplicate message.
254        // Remember info about the duplicate data to ignore future ack
255        s->data_merging = TRUE;
256        s->duplicate_gid = msg->last_gid;
257
258        // Send an ACK back to kill the duplicate
259        tw_stime ts;
260        INIT_TS_TRANSMIT(transmission_delay);
261        tw_event *e = genAckMessage(msg->last_gid, ts, lp, msg);
262        tw_event_send(e);
263
264        // update stats
265        s->total_transmit_time += ts;
266        s->messages_sent++;
267        printMessageInfo(lp->gid, msg->last_gid, tw_now(lp));
268      }
269    }
```

```
270 }
271
272 /**
273  * Processes the end of a sensor wait event.
274  * If the message is still saved forwards the message. Otherwise
275  * does nothing as an ack was received or the data was forwarded by another
276  * sensor.
277  *
278  * @param s The sensor associated with teh event.
279  * @param bf The time warp bit field for reverse computation.
280  * @param msg The sensor message for the event.
281  * @param lp The logical processor associated with this sensor.
282  */
283 void wait_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
284
285   // Only process the event if the events exactly match. The stillWaiting
286   // and remove functions only process the nonce and start_gid. Need to ensure
287   // that the path traveled is the same. This should remove duplications.
288   int location = stillWaiting(s, msg);
289
290   if (location >= 0) {
291 #ifdef VERBOSE
292   printf("%li WAIT test:location:%d   s_last:%G:m_last:%G   s_last_last:%G:m_last_la
293       lp->gid, location, s->wait_events[location]->last_gid, msg->last_gid,
294       s->wait_events[location]->last_last_gid, msg->last_last_gid);
295 #endif
296   }
297   if (location >= 0 && s->wait_events[location]->last_gid == msg->last_gid &&
298       s->wait_events[location]->last_last_gid == msg->last_last_gid &&
299       s->forwarding == NULL) {
300     remove_event(s, msg->source_gid, msg->nonce, lp);
301 #ifdef VERBOSE
302   printf("%li WAIT_END:%G\n", lp->gid, (double)tw_now(lp));
303 #endif
304
305     // Send an ACK to the previous gid. Indicates that this sensor
306     // self selected.
307     tw_stime ts;
308     INIT_TS_TRANSMIT(transmission_delay);
309     tw_event *e = genAckMessage(msg->last_gid, ts, lp, msg);
310     tw_event_send(e);
311
312     // Indicate to ourself that we forwarded the data.
313     s->forwarding = msg;
314
315     // update stats
316     s->total_transmit_time += ts;
317     s->messages_sent++;
318     printMessageInfo(lp->gid, msg->last_gid, tw_now(lp));
319
320     // Now send the data onward
321     msg->num_hops = s->sink_dist[0][0];
```

```
322       send_data(s, lp, msg);
323     } else {
324       // Another sensor self selected first.
325       // Remove the data
326       s->data_size -= msg->data_size;
327     }
328   }
329
330   /**
331    * Processes a data ack event.
332    * If we have forwarded this same message we are waiting for a neighbor to
333    * self select. Send out a DATA_ACK to neighbors.
334    * If the event is stored as waiting remove it as another sensor has forwarded
335    * the data.
336    *
337    * @param s The sensor associated with teh event.
338    * @param bf The time warp bit field for reverse computation.
339    * @param msg The sensor message for the event.
340    * @param lp The logical processor associated with this sensor.
341    */
342   void data_ack_event(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
343
344     // If we are waiting for a sensor to forward this message.
345     // Indicated by the stored forwarding message and whether this was
346     // the last_last_gid in the message
347     if (msg_equal(msg, s->forwarding) && msg->last_gid == lp->gid) {
348       // Indicate that we know someone else is forwarding
349       s->forwarding = NULL;
350       s->trying_to_send = FALSE;
351       s->data_size -= msg->data_size;
352       // send out an ack to neighbors.
353       sendMessage(s, lp, DATA_ACK, msg);
354
355       // Check to see if this sensor wants to sleep
356       if (s->wants_to_sleep == TRUE && s->num_wait_events == 0) {
357         go_to_sleep(s, bf, lp);
358       }
359
360     } else if (msg_equal(msg, s->forwarding)) {
361       // Do nothing. As this indicates an ack on data we are
362       // forwarding
363     } else if (stillWaiting(s, msg) >= 0) {
364       // Make sure that this is not a response to kill a duplicate.
365       if (s->data_merging == FALSE ||
366           (s->data_merging == TRUE && s->duplicate_gid != msg->last_gid)) {
367       remove_event(s, msg->source_gid, msg->nonce, lp);
368   #ifdef VERBOSE
369       printf("%li Another sensor selected\n", lp->gid);
370   #endif
371         // Check to see if this sensor wants to sleep
372         if (s->wants_to_sleep == TRUE && s->num_wait_events == 0 &&
373             s->forwarding == NULL) {
```

```
374          go_to_sleep(s, bf, lp);
375        }
376
377      } else {
378  #ifdef VERBOSE
379        printf("%li Duplicate killed\n", lp->gid);
380  #endif
381        s->data_merging = FALSE;
382        s->duplicate_gid = -1;
383      }
384    } else if (!(WHO_IS_SINK)){
385      printf("%li WARNING data duplication\n", lp->gid);
386    }
387
388  }
389
390
391
392
393
394
395
396
397  /**
398   * Reverse compiler event for initializing a sensor
399   *
400   * @param s The sensor associated with the event.
401   * @param bf The time warp bit field for reverse computation.
402   * @param msg The sensor message for the event.
403   * @param lp The logical processor with this sensor.
404   */
405  void sensor_init_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
406                      tw_lp *lp) {
407  }
408  /**
409   * Reverse compiler event for a sensor waking up
410   *
411   * @param s The sensor associated with the event.
412   * @param bf The time warp bit field for reverse computation.
413   * @param msg The sensor message for the event.
414   * @param lp The logical processor with this sensor.
415   */
416  void awake_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
417                      tw_lp *lp) {
418  }
419  /**
420   * Reverse compiler event for a sensor going to sleep
421   *
422   * @param s The sensor associated with the event.
423   * @param bf The time warp bit field for reverse computation.
424   * @param msg The sensor message for the event.
425   * @param lp The logical processor with this sensor.
```

```
426   */
427  void sleep_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
428                      tw_lp *lp) {
429  }
430  /**
431   * Reverse compiler event for a sink located
432   *
433   * @param s The sensor associated with the event.
434   * @param bf The time warp bit field for reverse computation.
435   * @param msg The sensor message for the event.
436   * @param lp The logical processor with this sensor.
437   */
438  void sink_loc_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
439                         tw_lp *lp) {
440    // set the state to forget the sink
441    remove_sink(s, msg->sink_gid);
442  }
443  /**
444   * Reverse compiler event for a sink lost
445   *
446   * @param s The sensor associated with the event.
447   * @param bf The time warp bit field for reverse computation.
448   * @param msg The sensor message for the event.
449   * @param lp The logical processor with this sensor.
450   */
451  void sink_lost_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
452                          tw_lp *lp) {
453
454  }
455  /**
456   * Reverse compiler event for data generation
457   *
458   * @param s The sensor associated with the event.
459   * @param bf The time warp bit field for reverse computation.
460   * @param msg The sensor message for the event.
461   * @param lp The logical processor with this sensor.
462   */
463  void data_gen_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
464                         tw_lp *lp) {
465    // remove the data
466    s->data_size -= data_size;
467    tw_rand_reverse_unif(lp->rng);
468  }
469  /**
470   * Reverse compiler event for data having been sent
471   *
472   * @param s The sensor associated with the event.
473   * @param bf The time warp bit field for reverse computation.
474   * @param msg The sensor message for the event.
475   * @param lp The logical processor with this sensor.
476   */
477  void data_send_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
```

```
478                         tw_lp *lp) {
479 }
480 /**
481  * Reverse compiler event for data being received
482  *
483  * @param s The sensor associated with the event.
484  * @param bf The time warp bit field for reverse computation.
485  * @param msg The sensor message for the event.
486  * @param lp The logical processor with this sensor.
487  */
488 void data_received_event_rc(sensor_state *s, tw_bf *bf,
489                             sensor_message *msg, tw_lp *lp) {
490 }
491 /**
492  * Reverse compiler event for data being received
493  *
494  * @param s The sensor associated with the event.
495  * @param bf The time warp bit field for reverse computation.
496  * @param msg The sensor message for the event.
497  * @param lp The logical processor with this sensor.
498  */
499 void wait_event_rc(sensor_state *s, tw_bf *bf,
500                             sensor_message *msg, tw_lp *lp) {
501 }
502 /**
503  * Reverse compiler event for data being received
504  *
505  * @param s The sensor associated with the event.
506  * @param bf The time warp bit field for reverse computation.
507  * @param msg The sensor message for the event.
508  * @param lp The logical processor with this sensor.
509  */
510 void expired_event_rc(sensor_state *s, tw_bf *bf,
511                             sensor_message *msg, tw_lp *lp) {
512 }
513 /**
514  * Reverse compiler event for data being received
515  *
516  * @param s The sensor associated with the event.
517  * @param bf The time warp bit field for reverse computation.
518  * @param msg The sensor message for the event.
519  * @param lp The logical processor with this sensor.
520  */
521 void data_forward_event_rc(sensor_state *s, tw_bf *bf,
522                             sensor_message *msg, tw_lp *lp) {
523 }
524 /**
525  * Reverse compiler event for data being received
526  *
527  * @param s The sensor associated with the event.
528  * @param bf The time warp bit field for reverse computation.
529  * @param msg The sensor message for the event.
```

```
530  * @param lp The logical processor with this sensor.
531  */
532 void data_ack_event_rc(sensor_state *s, tw_bf *bf,
533                              sensor_message *msg, tw_lp *lp) {
534 }
535 #endif
```

Listing A.9: Source Code

```
 1 /**
 2  * @file selfselect/messageUtils.h
 3  * @brief Messaging functions for use with the Self Select algorithm.
 4  * @author Thomas Reale
 5  * @date 12-2011
 6  * @version 1.0
 7  *
 8  * Functions to generate and send events to other sensors.
 9  *
10  */
11 #ifndef WSN_MESSAGE_UTILS_H
12 #define WSN_MESSAGE_UTILS_H
13
14 #include "../utils.h"
15
16
17 /** @def LAMBDA
18  * The lambda value used for calculating the backoff timer.
19  */
20 #define LAMBDA 10
21
22 /**
23  * Generates a Data Gen message
24  *
25  * @param lp The logical processor
26  *
27  * @return The created event.
28  */
29 tw_event *genDataGenMessage(tw_lp *lp) {
30   tw_stime ts = gdata_gen_rate;
31   tw_event *e;
32   sensor_message *m;
33
34   e = tw_event_new(lp->gid, ts, lp);
35   m = tw_event_data(e);
36   m->type = DATA_GEN;
37
38   return e;
39 }
40
41 /**
42  * Generates a sleep message to go to sleep in the future
43  *
44  * @param lp The logical processor
```

```
45  *
46  * @return The created event
47  */
48 tw_event *genSleepMessage(tw_lp *lp) {
49   tw_stime ts;
50
51   INIT_TS_NORMAL(gsleep_time);
52   sensor_message *m;
53   tw_event *e;
54
55   e = tw_event_new(lp->gid, ts, lp);
56   m = tw_event_data(e);
57   m->type = SLEEP;
58
59   return e;
60 }
61
62 /**
63  * Generates an awake message to wake up in the future
64  *
65  * @param lp The logical processor
66  *
67  * @return The created event
68  */
69 tw_event *genAwakeMessage(tw_lp *lp) {
70   tw_stime ts;
71   sensor_message *m;
72   tw_event *e;
73
74   INIT_TS_NORMAL(gawake_time);
75   e = tw_event_new(lp->gid, ts, lp);
76   m = tw_event_data(e);
77   m->type = AWAKE;
78
79   return e;
80 }
81
82 /**
83  * Generates a WAIT message for the backoff timer.
84  *
85  * @param s the snsor generating this wait message.
86  * @param lp The logical processor
87  * @param msg the message that triggered this wait.
88  *
89  * @return The created event
90  */
91 tw_event *genWaitMessage(sensor_state *s, tw_lp *lp, sensor_message *msg) {
92   tw_stime ts;
93
94   // Calculate the backoff timer.
95   double randValue = tw_rand_integer(lp->rng, 1, 20);
96   randValue = (randValue/20);
```

```
97     if (s->sink_dist[0][0] > msg->num_hops) {
98       ts = LAMBDA * (s->sink_dist[0][0] - msg->num_hops) * (randValue+1);
99     } else {
100      ts = LAMBDA * (randValue / (msg->num_hops - s->sink_dist[0][0] + 1));
101    }
102
103  #ifdef VERBOSE
104    printf("%li_timer:%G:msg_exp:%d:s_exp:%d:rand:%G:now:%G\n", lp->gid,
105           ts, msg->num_hops, s->sink_dist[0][0], randValue,
106           (double)tw_now(lp));
107  #endif
108    sensor_message *m;
109    tw_event *e;
110
111    e = tw_event_new(lp->gid, ts, lp);
112    m = tw_event_data(e);
113    copyMessageData(msg, m);
114    m->type = WAIT;
115
116    return e;
117  }
118
119  /**
120   * Sends a message to self only.
121   *
122   * @param s The state of the sensor.
123   * @param lp The logical processor associated with that sensor
124   * @param type The type of the message
125   * @param msg The message that triggered this send.
126   */
127  void sendSelfMessage(sensor_state *s, tw_lp *lp, sensor_event_t type,
128                       sensor_message *msg) {
129    tw_event *e = NULL;
130
131    switch(type) {
132      case SLEEP:
133        e = genSleepMessage(lp);
134        break;
135      case AWAKE:
136        e = genAwakeMessage(lp);
137        break;
138      case DATA_GEN:
139        e = genDataGenMessage(lp);
140        break;
141      case WAIT:
142        e = genWaitMessage(s, lp, msg);
143        // Save so another event can cancel and it can
144        // be checked.
145        save_event(s, e, lp);
146        break;
147      default:
148        break;
```

```
149     }
150
151     if (e != NULL) {
152       tw_event_send(e);
153     }
154
155   }
156
157   /**
158    * Generates a sinc_loc event
159    *
160    * @param s The sensor state containing information to put in message.
161    * @param dest The destination gid.
162    * @param ts How long in the future this event should be fired.
163    * @param lp This logical processor.
164    *
165    * @return The created sinc_loc event
166    */
167   tw_event *genSinkLocMessage(sensor_state *s, double dest, tw_stime ts,
168                               tw_lp *lp) {
169     tw_event *e;
170     sensor_message *m;
171
172     e = tw_event_new(dest, ts, lp);
173     m = tw_event_data(e);
174     m->type = SINK_LOC;
175     m->num_hops = s->sink_dist[0][0] +1;
176     m->last_gid = lp->gid;
177     m->sink_gid = s->sink_gid[0];
178     m->dest_gid = DEST_ALL;
179     return e;
180   }
181
182   /**
183    * Generates a sink lost event
184    *
185    * @param dest The destination gid
186    * @param ts How long in the future this event should be fired.
187    * @param lp This logical processor
188    *
189    * @return The created event
190    */
191   tw_event *genSinkLostMessage(double dest, tw_stime ts, tw_lp *lp) {
192     tw_event *e = NULL;
193     sensor_message *m;
194
195     e = tw_event_new(dest, ts, lp);
196     m = tw_event_data(e);
197     m->type = SINK_LOST;
198     m->dest_gid = DEST_ALL;
199     return e;
200   }
```

```
201 /**
202  * Generates a data received message at the given dest node.
203  *
204  * @param gid The GID of the destination node.
205  * @param ts How long in the future this event should be fired.
206  * @param lp The logical processor initiating the request.
207  * @param msg A message containing information about the data to send.
208  *
209  * @return the created event.
210  */
211 tw_event *genDataReceivedMessage(double gid, tw_stime ts, tw_lp *lp,
212                                  sensor_message *msg) {
213   sensor_message *m;
214   tw_event *e;
215
216   e = tw_event_new(gid, ts, lp);
217   m = tw_event_data(e);
218   copyMessageData(msg, m);
219   m->type = DATA_RECEIVED;
220   m->last_last_gid = m->last_gid;
221   m->last_gid = lp->gid;
222
223   return e;
224 }
225
226 /**
227  * Generates an ACK message at the given dest node.
228  *
229  * @param gid The GID of the destination node.
230  * @param ts How long in the future this event should be fired.
231  * @param lp The logical processor initiating the request.
232  * @param msg A message containing information about the ACK.
233  *
234  * @return The created event.
235  */
236 tw_event *genAckMessage(double gid, tw_stime ts, tw_lp *lp,
237                         sensor_message *msg) {
238   sensor_message *m;
239   tw_event *e;
240
241   e = tw_event_new(gid, ts, lp);
242   m = tw_event_data(e);
243   copyMessageData(msg, m);
244   m->type = DATA_ACK;
245   // Do not update the last/last_last gids.
246   // This is necessary to identify at what stage the message is in.
247   return e;
248 }
249
250 /**
251  * Sends a message originating from s to all nearby sensors.
252  *
```

```
253  * @param s The originating sensor
254  * @param lp the associated lp.
255  * @param type The event type to send
256  * @param msg The message that triggered this send.
257  */
258 void sendMessage(sensor_state *s, tw_lp *lp, sensor_event_t type,
259                  sensor_message *msg) {
260   //sensor_message *m;
261   tw_event *e = NULL;
262
263   // Take care of events that alter the state of the originating sensor
264   switch (type) {
265     case SINK_DISCOVERED:
266       add_sink(s, lp->gid, lp->gid, 0, sensor_power);
267       type = SINK_LOC;
268       break;
269     default:
270       break;
271   }
272
273   // Calculate how long the message will take.
274   tw_stime ts;
275   if (type == DATA_RECEIVED) {
276     INIT_TS_TRANSMIT(msg->data_size / gtransmit_rate);
277   } else {
278     INIT_TS_TRANSMIT(transmission_delay);
279   }
280
281   s->power -= power_loss(ts, transmit_power);
282
283   // Send the requested message if valid type to all nearby neighbors
284   int i = 0;
285   for (i = 0; i < s->num_nearby; ++i) {
286     switch (type) {
287       case SINK_LOC:
288         e = genSinkLocMessage(s, s->nearby_Sensors[i], ts, lp);
289         break;
290       case SINK_LOST:
291         e = genSinkLostMessage(s->nearby_Sensors[i], ts, lp);
292         break;
293       case DATA_RECEIVED:
294         e = genDataReceivedMessage(s->nearby_Sensors[i], ts, lp, msg);
295         break;
296       case DATA_ACK:
297         e = genAckMessage(s->nearby_Sensors[i], ts, lp, msg);
298         break;
299       default:
300         // These types should not be associated with this type
301         // of message sending.
302         break;
303     }
304     if (e != NULL) {
```

```
305         tw_event_send(e);
306         printMessageInfo(lp->gid, s->nearby_Sensors[i], tw_now(lp));
307         // The multiple messages are used for the simulation. In reality
308         // this would be done by a single transmission.
309         s->messages_sent++;
310         s->total_transmit_time += ts;
311
312       } else {
313         printf("WARNING: Event Not Sent\n");
314       }
315     }
316 }
317
318 /**
319  * Attempts to send data to a known sink.
320  * If no sink is known, broadcasts a SINK_LOST message to update all around.
321  *
322  * @param s The sensor state.
323  * @param lp The processor associated with the lp
324  * @param msg the message that triggered this send data.
325  */
326 void send_data(sensor_state *s, tw_lp *lp, sensor_message *msg) {
327   if (s->trying_to_send == FALSE && s->sink_gid[0] != UNKNOWN &&
328       s->data_size > 0) {
329
330     double data_size = s->data_size;
331     int isNew = 0;
332     if (msg == NULL) {
333       msg = (sensor_message *)malloc(sizeof(sensor_message));
334
335       msg->source_gid = lp->gid;
336       msg->nonce = s->messages_sent;
337       msg->data_size = data_size;
338       msg->start_time = tw_now(lp);
339       msg->last_gid = lp->gid;
340       msg->last_last_gid = lp->gid;
341       msg->num_hops = s->sink_dist[0][0]-1;
342       isNew = 1;
343       s->forwarding = msg;
344     } else if (msg->source_gid == lp->gid) {
345       // If this is the original source, combine all data possible.
346       msg->data_size = data_size;
347     }
348
349     // Indicate we are sending data
350     s->trying_to_send = TRUE;
351
352     // Send data to all neighbors
353     sendMessage(s, lp, DATA_RECEIVED, msg);
354
355     // Free up memory
356     if (isNew == 1) {
```

```
357         free(msg);
358         msg = NULL;
359       }
360     }
361   }
362
363   #endif
```

Listing A.10: Source Code

```
1   /**
2    * @file selfselect/wsn.h
3    * @brief Defines the structures and any Self Select specific variables.
4    * @author Thomas Reale
5    * @date 12-2011
6    * @version 1.0
7    *
8    * Defines the message structure, the sensor structure, and possible events.
9    *
10   */
11  #ifndef INC_wsn_h
12  #define INC_wsn_h
13
14  #include <ross.h>
15
16  /** typedef to identify an event */
17  typedef enum sensor_event_t sensor_event_t;
18  /** typedef to define a sensor's state */
19  typedef struct sensor_state sensor_state;
20  /** typedef to define a message between sensors */
21  typedef struct sensor_message sensor_message;
22  /** typedef to define a location */
23  typedef struct Location Location;
24  /** typedef to define a sensor structure */
25  typedef struct Sensor Sensor;
26
27  /** @def MAX_SINKS
28   * Used to set the maximum number of sinks stored */
29  #define MAX_SINKS 5
30
31  /** @def MAX_PATHS_PER_SINK
32   * Defines the maximum number of alternate paths to each sink */
33  #define MAX_PATHS_PER_SINK 2
34
35  /** @def MAX_WAIT
36   * The maximum number of messages a single sensor will be able to store */
37  #define MAX_WAIT 10
38
39  /** Keeps track of all sensors to allow to simulate distance between sensors */
40  static Sensor **sensors_list;
41
42
43  /** The possible events associated with a sensor */
```

```
44 enum sensor_event_t
45 {
46   SLEEP = 1,
47   AWAKE,
48   DEAD,
49   SINK_LOC,
50   SINK_LOST,
51   SINK_DISCOVERED,
52   DATA_GEN,
53   DATA_ACK,
54   DATA_RECEIVED,
55   INIT,
56   WAIT
57 };
58
59 /** contains a cartesian coordinate */
60 struct Location {
61   /** The horizontal position of a sensor */
62   long x;
63   /** The vertical position of a sensor */
64   long y;
65 };
66
67 /** Information about a sensor for others to use for communication */
68 struct Sensor {
69   /** The location of this sensor */
70   Location location;
71   /** The global ID of this sensor */
72   long gid;
73 };
74 /**
75  * Contains the information related to a single sensor
76  * including statistics.
77  */
78 struct sensor_state
79 {
80   /** Power remaining in the sensor */
81   double power;
82   /** The state of the sensor */
83   int state;
84   /** The type of the sensor (SINK,NODE=1) */
85   int type;
86   /** Location of the sensor */
87   Location location;
88   /** The data size on the device in bytes */
89   double data_size;
90   /** The GID to communicate with in order to send data to a sink.
91    * corresponds with sink_gid. Paths are ordered according to dist
92    */
93   double sink_path[MAX_SINKS][MAX_PATHS_PER_SINK];
94   /** The distance to the data sink. 0 indicates either that
95    *  this is the sink a negative value indicates no sink.
```

```
96      *  corresponds with sink_gid
97      */
98    int sink_dist[MAX_SINKS][MAX_PATHS_PER_SINK];
99    /** The ID of the sink, to stop continuous broadcasts. Array is ordered
100     * as inserting and deleting is less frequent than accessing */
101   double sink_gid[MAX_SINKS];
102   /** A list of the GIDs for nearby sensors. */
103   long *nearby_Sensors;
104   /** The number of nearby sensors */
105   int num_nearby;
106   /** Events that need to be recorded for future knowledge */
107   sensor_message *wait_events[MAX_WAIT];
108   /** The number of wait events in the queue */
109   int num_wait_events;
110   /** Events that this sensor is attempting to forward */
111   sensor_message *forwarding;
112   /** Indicates that this sensor is sending data */
113   int trying_to_send;
114   /** Indicates that data is merging. */
115   int data_merging;
116   /** Indicates who the duplicate GID is for merging data */
117   int duplicate_gid;
118
119   /** Indicator to indicate that this sensor wants to sleep when possible */
120   int wants_to_sleep;
121   /** Last time the sensor's state changed for stats purposes */
122   tw_stime last_state_change;
123
124   // Used for statistics
125   /** Total number of times awake */
126   double times_awake;
127   /** Total number of times asleep */
128   double times_asleep;
129   /** Total number of messages sent */
130   double messages_sent;
131   /** Total awake time */
132   tw_stime total_awake_time;
133   /** Total asleep time */
134   tw_stime total_asleep_time;
135   /** Total transmit time */
136   tw_stime total_transmit_time;
137 };
138
139 /**
140  * A message to trigger events
141  */
142 struct sensor_message
143 {
144   /** The type of event this message is associated with */
145   sensor_event_t type;
146   /** The data size contained in this message (DATA RECEIVED/DATA GEN) */
147   double data_size;
```

```
148     /** The original source of the message */
149     double source_gid;
150     /** The destination gid of this message or DEST_ALL */
151     double dest_gid;
152     /** The power the last sensor had */
153     double last_power;
154 
155     // Fields for messages originating with sinks In SINK_LOC message,
156     /** Number of hops to sink */
157     int num_hops;
158     /** In SINK_LOC message, gid of last location on path to sink. */
159     double last_gid;
160     /** The sensor before the last sensor */
161     double last_last_gid;
162     /** The gid of the sink */
163     double sink_gid;
164     /** A nonce to identify this message */
165     int nonce;
166 
167     // statistics variables
168     tw_stime start_time;
169   };
170 
171   #endif
```

A.5 KAS Files

Listing A.11: Source Code

```
 1  /**
 2   * @file assisted/assisted.h
 3   * @brief Processes events for the KASS algorithm.
 4   * @author Thomas Reale
 5   * @date 12-2011
 6   * @version 1.0
 7   *
 8   * Implements the KASS algorithm by processing all events that occur
 9   * during the simulation.
10   */
11 
12  #ifndef WSN_ASSISTED
13  #define WSN_ASSISTED
14 
15  #include "wsn.h"
16  #include "../utils.h"
17  #include "messageUtils.h"
18 
19  /** @def WHO_IS_SINK
20   * Who is the sink based on an lp's gid */
21  #define WHO_IS_SINK (lp->gid==1)
22  /** @def WHO_GEN_DATA
23   * Who generates data based on lp's gid */
24  #define WHO_GEN_DATA (lp->gid==0)
```

```
25
26 /**
27  * Initializes all variables in a single sensor.
28  *
29  * @param s The sensor to initialize.
30  */
31 void initializeVariables(sensor_state *s) {
32   s->nearby_Sensors = NULL;
33   s->num_nearby = 0;
34   s->state = AWAKE;
35   s->num_wait_events = 0;
36   s->trying_to_send = 0;
37   s->wants_to_sleep = FALSE;
38   s->power = sensor_power;
39
40   // statistics
41   s->times_awake = 0;
42   s->times_asleep = 0;
43   s->messages_sent = 0;
44   s->total_awake_time = 0;
45   s->total_asleep_time = 0;
46 }
47
48 /**
49  * The events that will occur when the INIT event is trigered.
50  * This event is called once at the very beginning of the simulation.
51  *
52  * Sets up which sensors identify as sinks, which sensors generate data
53  * as well as initiating awake/sleep cycle. (currently doesn't actually wake/sleep c
54  *
55  * @param s The sensor associated with this event
56  * @param bf The time warp bit field for reverse computation.
57  * @param msg The sensor message for the event.
58  * @param lp The logical processor with the sensor
59  */
60 void sensor_init(sensor_state *s, tw_bf *bf, sensor_message *msg, tw_lp *lp) {
61   findNearbySensors(s, lp->gid);
62
63   if (WHO_IS_SINK) {
64     sendMessage(s, lp, SINK_DISCOVERED,NULL);
65   }
66
67   if (WHO_GEN_DATA) {
68     // start collecting data
69     sendSelfMessage(s, lp, DATA_GEN, NULL);
70   }
71
72   // All but the sink sleep
73   if (!(WHO_IS_SINK)) {
74     // Start the Awake Sleep Cycle
75     s->last_state_change = 0;
76     sendSelfMessage(s, lp, SLEEP, NULL);
```

```
77     }
78   }
79
80   /**
81    * Processes an awake event on a sensor.
82    * The sensor enters the awake state and schedules a sleep event.
83    *
84    * @param s The sensor associated with the event
85    * @param bf The time warp bitfield.
86    * @param msg the message associated with this event.
87    * @param lp The logical processor with this sensor
88    */
89   void awake_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
90                    tw_lp *lp) {
91
92     // Update Stats
93     double timeSpent = tw_now(lp) - s->last_state_change;
94     s->total_asleep_time += timeSpent;
95     s->times_asleep++;
96     s->power -= power_loss(timeSpent, sleep_power);
97     // Change state to AWAKE
98     s->state = AWAKE;
99     s->last_state_change = tw_now(lp);
100
101    // Create an event to go to sleep
102    sendSelfMessage(s, lp, SLEEP, NULL);
103  }
104
105  /**
106   * Sets the sensor to a sleep state and schedules an AWAKE event in the future.
107   *
108   * @param s The sensor going to sleep.
109   * @param bf The TW bf for RC.
110   * @param lp The logical processor associated with this sensor.
111   */
112  void go_to_sleep(sensor_state *s, tw_bf *bf, tw_lp *lp) {
113
114    s->state = SLEEP;
115    s->wants_to_sleep = FALSE;
116    // Update Stats
117    double timeSpent = tw_now(lp) - s->last_state_change;
118    s->total_awake_time += timeSpent;
119    s->times_awake++;
120    s->power -= power_loss(timeSpent, active_power);
121
122    // Create an event to wake up
123    s->last_state_change = tw_now(lp);
124    sendSelfMessage(s, lp, AWAKE, NULL);
125  }
126
127  /**
128   * Processes a sleep event on a sensor.
```

```
129   * If the sensor is not processing a message goes to sleep.
130   * Otherwise sets the wants_to_sleep variable to true in the sensor.
131   *
132   * @param s The sensor associated with the event
133   * @param bf The time warp bit field.
134   * @param msg The message associated with this event.
135   * @param lp The logical processor with this sensor
136   */
137  void sleep_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
138                   tw_lp *lp) {
139    // Check to see if it is possible to go to sleep.
140    // Make sure not waiting to forward data
141    if (s->num_wait_events == 0) {
142      go_to_sleep(s, bf, lp);
143    } else {
144      s->wants_to_sleep = TRUE;
145  #ifdef VERBOSE
146      if (WHO_GEN_DATA) {
147        printf("%li Wants to sleep\n", lp->gid);
148      }
149  #endif
150    }
151  }
152
153  /**
154   * Processes a sink loc event on a sensor
155   * Stores the sink on the sensor and broadcasts SINK_LOC to neighbors.
156   *
157   * @param s The sensor associated with the event
158   * @param bf The time warp bit field.
159   * @param msg The message received
160   * @param lp The logical processor with this sensor
161   */
162  void sink_loc_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
163                      tw_lp *lp) {
164    // Indicates a sink is broadcasting it's position
165    if (add_sink(s, msg->sink_gid, msg->last_gid, msg->num_hops,
166          msg->last_power)) {
167      // Broadcast the message to all surrounding sensors
168      sendMessage(s, lp, SINK_LOC,NULL);
169    }
170
171    // start sending data (if any) to the sink
172    send_data(s, lp, NULL);
173  }
174
175  /**
176   * Processes a sink lost event on a sensor
177   * Removes the sink from the sensor and broadcasts SINK_LOST to neighbors.
178   *
179   * @param s The sensor associated with the event
180   * @param bf The time warp bit field.
```

```
181   * @param msg The message associated with this event
182   * @param lp The logical processor with this sensor
183   */
184  void sink_lost_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
185                       tw_lp *lp) {
186    if (remove_sink(s, msg->sink_gid)) {
187      sendMessage(s, lp, SINK_LOST,NULL);
188    }
189  }
190
191  /**
192   * Processes a data gen event on a sensor
193   * Adds data to the sensors storage and starts sending data.
194   * If this sensor is the sink just updates statistics indicating
195   * completed reception. Schedules a DATA_GEN in the future.
196   *
197   * @param s The sensor associated with the event
198   * @param bf The time warp bit field.
199   * @param msg the message assicated with this event.
200   * @param lp The logical processor with this sensor
201   */
202  void data_gen_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
203                      tw_lp *lp) {
204    // Only create data and send it if we are not the sink
205    if (!(WHO_IS_SINK)) {
206      // update state
207      s->data_size += data_size;
208
209      // Send data if possible
210      send_data(s, lp, NULL);
211    } else {
212      // update statistics
213      transmission_recv_ct++;
214      data_sink_received += data_size;
215    }
216    // generate data in the future
217    sendSelfMessage(s, lp, DATA_GEN, NULL);
218    printStateInfo(lp->gid, tw_now(lp), "DATA", s->data_size, s->state);
219  }
220
221  /**
222   * Processes a data received event on a sensor
223   * If the message indicates this is the destination forwards the message on.
224   * If this is not the destination starts a DATA_EXPIRED message to self indicating
225   * when the data should expire.
226   *
227   * @param s The sensor associated with the event
228   * @param bf The time warp bit field.
229   * @param msg The message associated with this event.
230   * @param lp The logical processor with this sensor
231   */
232  void data_received_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
```

```
233                              tw_lp *lp) {
234 #ifdef VERBOSE
235   double time = tw_now(lp);
236   printf("%li data_received.\tmsg_nonce:%d\tmsg_source:%G\t"
237       "msg_nearby:%g:data_size:%G:status:%d:%G\n",
238       lp->gid, msg->nonce, msg->source_gid, msg->num_nearby, msg->data_size,
239       s->state, time);
240 #endif
241
242   // If this is the destination, send an ack back to the originator and
243   // forward the data.
244   s->data_size += msg->data_size;
245
246   //update the list of neighbors according to the power received.
247   //update_cost(s, msg->sink_gid, msg->last_gid, msg->num_hops,
248   //            msg->last_power);
249
250   if (msg->dest_gid == lp->gid) {
251     forward_data(s, lp, msg);
252   } else {
253     // else this is not the direct destination.
254     // Start a backoff timer
255     sendSelfMessage(s, lp, DATA_EXPIRED, msg);
256   }
257 }
258
259 /**
260  * Processes a data expiration. When this is not the sensor that
261  * forwards the data, this sensor is holding the data until it decides
262  * that another sensor has forwarded the data. At this time, this sensor
263  * can free up the data.
264  *
265  * @param s The sensor associated with the event
266  * @param bf The time warp bit field.
267  * @param msg The message received.
268  * @param lp The logical processor with this sensor
269  */
270 void expired_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
271                    tw_lp *lp) {
272 #ifdef VERBOSE
273   printf("%li Data Expired:%G\n", lp->gid, tw_now(lp));
274 #endif
275   // remove the event from the saved list
276   if (remove_event(s, msg->source_gid, msg->nonce, lp)) {
277     // If the event was still there then the data has been forwarded.
278     s->data_size -= msg->data_size;
279     // Check to see if this sensor can now go to sleep if it wants to.
280     if (s->wants_to_sleep == TRUE && s->num_wait_events == 0) {
281       go_to_sleep(s, bf, lp);
282     }
283   }
284 }
```

```
285
286 /**
287  * Processes a data ack event on a sensor
288  * Removes the message from being stored that is associated with the message
289  * passed in. If the sensor wants to sleep starts sleeping.
290  *
291  * @param s The sensor associated with teh event
292  * @param bf The time warp bit field.
293  * @param msg The message received.
294  * @param lp The logical processor with this sensor
295  */
296 void data_ack_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
297                     tw_lp *lp) {
298
299 #ifdef VERBOSE
300   printf("%li ACK received:source:%G:nonce:%d:data:%G:%G\n",
301       lp->gid, msg->source_gid, msg->nonce, msg->data_size, tw_now(lp));
302 #endif
303   // Indicates that a neighbor self selected.
304   // remove this message from the waiting so the wait ignores it.
305   remove_event(s, msg->source_gid, msg->nonce, lp);
306   // Now the data is guaranteed on the other sensor, so remove it
307   // from this sensor.
308   s->data_size -= msg->data_size;
309   s->trying_to_send = FALSE;
310
311   // Check to see if this sensor wants to and can go to sleep.
312   if (s->wants_to_sleep == TRUE && s->num_wait_events == 0) {
313     go_to_sleep(s, bf, lp);
314   }
315 }
316
317 /**
318  * An event indicating that the sensor has been waiting for a sensor
319  * to forward the data has expired. Send a data forward event to the
320  * next best sensor.
321  *
322  * @param s The sensor
323  * @param bf The time warp bit field.
324  * @param msg the message associated with this event.
325  * @param lp The logical processor
326  */
327 void wait_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
328                 tw_lp *lp) {
329
330   // If an ack was not received and if this sensor has not recently
331   // handled the data.
332   if (msg->last_gid != lp->gid &&
333       remove_event(s, msg->source_gid, msg->nonce, lp) >= 0) {
334     // Find the next best sensor to forward the data.
335     int foundCurrent = 0;
336     long newDest = UNKNOWN;
```

```
337       int i = 0;
338       int j = 0;
339       for (i = 0; i < MAX_SINKS; ++i) {
340         for (j = 0; j < MAX_PATHS_PER_SINK; ++j) {
341           if (foundCurrent == 1 && s->sink_path[i][j] != UNKNOWN) {
342             newDest = s->sink_path[i][j];
343             break;
344           } else if (s->sink_path[i][j] == msg->dest_gid) {
345               // find the next available path
346               foundCurrent = 1;
347           }
348         }
349       }
350
351 #ifdef VERBOSE
352         double time = tw_now(lp);
353         printf("%li WAIT_EXPIRED src:%G:nonce:%d:dest: %lf:newdest:%li:%G\n",
354               lp->gid, msg->source_gid, msg->nonce, msg->dest_gid,
355                newDest, time);
356 #endif
357       // If we have checked all paths, re-send the data to see if nearby
358       // sensors have woken up
359       if (newDest == UNKNOWN) {
360 #ifdef VERBOSE
361         printf("%li All Neighbors Tested. Resending\n", lp->gid);
362 #endif
363         s->trying_to_send = 0;
364         send_data(s, lp, msg);
365         return;
366       } else {
367         // Else send a DATA_FORWARD event to the next best choice.
368         msg->dest_gid = newDest;
369         tw_stime ts = transmission_delay;
370         tw_event *e = genDataForwardMessage(lp, msg, ts);
371         tw_event_send(e);
372
373         printMessageInfo(lp->gid, msg->dest_gid, tw_now(lp));
374
375         // create another wait event.
376         sendSelfMessage(s, lp, WAIT, msg);
377
378         // update stats
379         s->total_transmit_time += ts;
380       }
381     }
382 }
383
384 /**
385  * An event indicating that data we contain should be forwarded.
386  *
387  * @param s The sensor contianing the data.
388  * @param bf The time warp bitfield.
```

```
389  * @param msg The forward message request.
390  * @param lp the logical processor associated with the sensor.
391  */
392 void data_forward_event(sensor_state *s, tw_bf *bf, sensor_message *msg,
393                          tw_lp *lp) {
394
395 #ifdef VERBOSE
396   printf("%li DATA_FORWARD receieved:%G\n", lp->gid, tw_now(lp));
397 #endif
398
399   // Remove the message from the saved messages
400   remove_event(s, msg->source_gid, msg->nonce, lp);
401
402   // Forward the data
403   forward_data(s, lp, msg);
404 }
405
406
407
408
409
410
411
412
413
414
415
416 /**
417  * Reverse compiler event for initializing a sensor
418  *
419  * @param s The sensor associated with the event.
420  * @param bf The time warp bitfield.
421  * @param msg the message associated with this event.
422  * @param lp The logical processor with this sensor.
423  */
424 void sensor_init_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
425                     tw_lp *lp) {
426 }
427 /**
428  * Reverse compiler event for a sensor waking up
429  *
430  * @param s The sensor associated with the event.
431  * @param bf The time warp bitfield.
432  * @param msg the message associated with this event.
433  * @param lp The logical processor with this sensor.
434  */
435 void awake_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
436                     tw_lp *lp) {
437 }
438 /**
439  * Reverse compiler event for a sensor going to sleep
440  *
```

```
441   * @param s The sensor associated with the event.
442   * @param bf The time warp bitfield.
443   * @param msg the message associated with this event.
444   * @param lp The logical processor with this sensor.
445   */
446  void sleep_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
447                      tw_lp *lp) {
448  }
449  /**
450   * Reverse compiler event for a sink located
451   *
452   * @param s The sensor associated with the event.
453   * @param bf The time warp bitfield.
454   * @param msg the message associated with this event.
455   * @param lp The logical processor with this sensor.
456   */
457  void sink_loc_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
458                          tw_lp *lp) {
459    // set the state to forget the sink
460    remove_sink(s, msg->sink_gid);
461  }
462  /**
463   * Reverse compiler event for a sink lost
464   *
465   * @param s The sensor associated with the event.
466   * @param bf The time warp bitfield.
467   * @param msg the message associated with this event.
468   * @param lp The logical processor with this sensor.
469   */
470  void sink_lost_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
471                          tw_lp *lp) {
472
473  }
474  /**
475   * Reverse compiler event for data generation
476   *
477   * @param s The sensor associated with the event.
478   * @param bf The time warp bitfield.
479   * @param msg the message associated with this event.
480   * @param lp The logical processor with this sensor.
481   */
482  void data_gen_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
483                          tw_lp *lp) {
484    // remove the data
485    s->data_size -= data_size;
486    tw_rand_reverse_unif(lp->rng);
487  }
488  /**
489   * Reverse compiler event for data having been sent
490   *
491   * @param s The sensor associated with the event.
492   * @param bf The time warp bitfield.
```

```
493  * @param msg the message associated with this event.
494  * @param lp The logical processor with this sensor.
495  */
496 void data_send_event_rc(sensor_state *s, tw_bf *bf, sensor_message *msg,
497                         tw_lp *lp) {
498 }
499 /**
500  * Reverse compiler event for data being received
501  *
502  * @param s The sensor associated with the event.
503  * @param bf The time warp bitfield.
504  * @param msg the message associated with this event.
505  * @param lp The logical processor with this sensor.
506  */
507 void data_received_event_rc(sensor_state *s, tw_bf *bf,
508                             sensor_message *msg, tw_lp *lp) {
509 }
510 /**
511  * Reverse compiler event for data being received
512  *
513  * @param s The sensor associated with the event.
514  * @param bf The time warp bitfield.
515  * @param msg the message associated with this event.
516  * @param lp The logical processor with this sensor.
517  */
518 void wait_event_rc(sensor_state *s, tw_bf *bf,
519                             sensor_message *msg, tw_lp *lp) {
520 }
521
522 /**
523  * Reverse compiler event for data being received
524  *
525  * @param s The sensor associated with the event.
526  * @param bf The time warp bitfield.
527  * @param msg the message associated with this event.
528  * @param lp The logical processor with this sensor.
529  */
530 void expired_event_rc(sensor_state *s, tw_bf *bf,
531                             sensor_message *msg, tw_lp *lp) {
532 }
533 /**
534  * Reverse compiler event for data being received
535  *
536  * @param s The sensor associated with the event.
537  * @param bf The time warp bitfield.
538  * @param msg the message associated with this event.
539  * @param lp The logical processor with this sensor.
540  */
541 void data_forward_event_rc(sensor_state *s, tw_bf *bf,
542                             sensor_message *msg, tw_lp *lp) {
543 }
544 /**
```

```
545  * Reverse compiler event for data being received
546  *
547  * @param s The sensor associated with the event.
548  * @param bf The time warp bitfield.
549  * @param msg the message associated with this event.
550  * @param lp The logical processor with this sensor.
551  */
552 void data_ack_event_rc(sensor_state *s, tw_bf *bf,
553                              sensor_message *msg, tw_lp *lp) {
554 }
555
556 #endif
```

Listing A.12: Source Code

```
1  /**
2   * @file assisted/messageUtils.h
3   * @brief Messaging functions for use with the KASS algorithm.
4   * @author Thomas Reale
5   * @date 12-2011
6   * @version 1.0
7   *
8   * Functions to generate and send events to other sensors.
9   *
10  */
11
12 #ifndef WSN_MESSAGE_UTILS
13 #define WSN_MESSAGE_UTILS
14
15 #include "../utils.h"
16
17
18 /**
19  * Generates a sleep message to go to sleep in the future
20  *
21  * @param lp The logical processor
22  *
23  * @return The created event
24  */
25 tw_event *genSleepMessage(tw_lp *lp) {
26   tw_stime ts;
27   tw_event *e;
28   sensor_message *m;
29
30   INIT_TS_NORMAL(gsleep_time);
31   e = tw_event_new(lp->gid, ts, lp);
32   m = tw_event_data(e);
33   m->type = SLEEP;
34   m->start_time = tw_now(lp);
35
36   return e;
37 }
38
```

```
39  /**
40   * Generates an awake message to wake up in the future
41   *
42   * @param lp The logical processor
43   *
44   * @return The created event
45   */
46  tw_event *genAwakeMessage(tw_lp *lp) {
47    tw_stime ts;
48    tw_event *e;
49    sensor_message *m;
50
51    INIT_TS_NORMAL(gawake_time);
52    e = tw_event_new(lp->gid, ts, lp);
53    m = tw_event_data(e);
54    m->type = AWAKE;
55    m->start_time = tw_now(lp);
56
57    return e;
58  }
59
60  /**
61   * Generates a Data Gen message
62   *
63   * @param lp The logical processor
64   *
65   * @return The created event.
66   */
67  tw_event *genDataGenMessage(tw_lp *lp) {
68    tw_stime ts = gdata_gen_rate;
69    tw_event *e;
70    sensor_message *m;
71
72    e = tw_event_new(lp->gid, ts, lp);
73    m = tw_event_data(e);
74    m->type = DATA_GEN;
75
76    return e;
77  }
78
79  /**
80   * Generates a data expired message
81   *
82   * The data will expire after waiting for
83   * 4 * transmission_delay * sensors_nearby_sender
84   *
85   * @param lp the logical processor.
86   * @param msg The message causing this timer to start
87   *
88   * @return The created event
89   */
90  tw_event *genDataExpiredMessage(tw_lp *lp, sensor_message *msg) {
```

```
91    tw_stime ts = msg->num_nearby * transmission_delay * 4;
92    tw_event *e;
93    sensor_message *m;
94
95    e = tw_event_new(lp->gid, ts, lp);
96    m = tw_event_data(e);
97    copyMessageData(msg, m);
98    m->type = DATA_EXPIRED;
99
100   return e;
101 }
102
103
104 /**
105  * Generates a wait event to wait for a device to forward data.
106  *
107  *  The amount of time to wait is 3*transmission_delay. In that time
108  *  it must receive an ack to confirm that the data is being forwarded.
109  *
110  * @param s The sensor state.
111  * @param lp The logical processor
112  * @param msg The message that triggered a wait. Used for backoff timer.
113  *
114  * @return An event that indicates the wait time has completed.
115  */
116 tw_event *genWaitMessage(sensor_state *s, tw_lp *lp, sensor_message *msg) {
117   tw_stime ts = 4*transmission_delay;
118   tw_event *e;
119   sensor_message *m;
120
121   e = tw_event_new(lp->gid, ts, lp);
122   m = tw_event_data(e);
123   copyMessageData(msg, m);
124
125   m->type = WAIT;
126
127   return e;
128 }
129
130 /**
131  * Sends a message to self only.
132  *
133  * @param s The state of the sensor.
134  * @param lp The logical processor associated with that sensor
135  * @param type The type of the message
136  * @param msg The message that triggered this action.
137  */
138 void sendSelfMessage(sensor_state *s, tw_lp *lp, sensor_event_t type,
139                      sensor_message *msg) {
140   tw_event *e = NULL;
141
142   switch(type) {
```

```
143       case SLEEP:
144         e = genSleepMessage(lp);
145         break;
146       case AWAKE:
147         e = genAwakeMessage(lp);
148         break;
149       case DATA_GEN:
150         e = genDataGenMessage(lp);
151         break;
152       case DATA_EXPIRED:
153         e = genDataExpiredMessage(lp, msg);
154         // this event must also be saved
155         save_event(s, e, lp);
156         break;
157       case WAIT:
158         e = genWaitMessage(s, lp, msg);
159         // this event must be saved
160         save_event(s, e, lp);
161         break;
162       default:
163         break;
164     }
165
166     if (e != NULL) {
167       tw_event_send(e);
168     }
169
170   }
171
172   /**
173    * Generates an ack message to indicate to the previous sensor that
174    * this sensor can forward data.
175    * @param lp the logical processor.
176    * @param msg The data received message causing this ack to be sent.
177    *
178    * @return the created event.
179    */
180   tw_event *genAckEvent(tw_lp *lp, sensor_message *msg) {
181     tw_stime ts;
182
183     INIT_TS_TRANSMIT(transmission_delay);
184     tw_event *e;
185     sensor_message *m;
186
187     e = tw_event_new(msg->last_gid, ts, lp);
188     m = tw_event_data(e);
189     copyMessageData(msg, m);
190
191     m->type = DATA_ACK;
192     m->dest_gid = msg->last_gid;
193     m->last_last_gid = msg->last_gid;
194     m->last_gid = lp->gid;
```

```
195
196     return e;
197 }
198
199 /**
200  * Generates a data forward message to tell a sensor
201  * to forward data.
202  *
203  * @param lp the logical processor.
204  * @param msg A message containing information about the
205  *             data to forward.
206  * @param ts How far in the future this event should fire.
207  *
208  * @return The created event
209  */
210 tw_event *genDataForwardMessage(tw_lp *lp, sensor_message *msg,
211                                 tw_stime ts) {
212     tw_event *e;
213     sensor_message *m;
214
215     e = tw_event_new(msg->dest_gid, ts, lp);
216     m = tw_event_data(e);
217     copyMessageData(msg, m);
218
219     m->type = DATA_FORWARD;
220
221     return e;
222 }
223
224 /**
225  * Generates a sinc_loc event
226  *
227  * @param s The sensor state containing information to put in message.
228  * @param dest The destination gid.
229  * @param lp This logical processor.
230  * @param ts How far in the future this event should fire.
231  *
232  * @return The created sinc_loc event
233  */
234 tw_event *genSinkLocMessage(sensor_state *s, double dest, tw_lp *lp,
235                             tw_stime ts) {
236     tw_event *e;
237     sensor_message *m;
238
239     e = tw_event_new(dest, ts, lp);
240     m = tw_event_data(e);
241     m->type = SINK_LOC;
242     m->num_hops = s->sink_dist[0][0] +1;
243     m->last_gid = lp->gid;
244     m->sink_gid = s->sink_gid[0];
245     m->last_power = s->power;
246
```

```
247   return e;
248 }
249
250 /**
251  * Generates a sink lost event
252  *
253  * @param dest The destination gid
254  * @param lp This logical processor
255  * @param ts How far in the future this event should fire.
256  *
257  * @return The created event
258  */
259 tw_event *genSinkLostMessage(double dest, tw_lp *lp, tw_stime ts) {
260   tw_event *e = NULL;
261   sensor_message *m;
262
263   e = tw_event_new(dest, ts, lp);
264   m = tw_event_data(e);
265   m->type = SINK_LOST;
266   return e;
267 }
268
269 /**
270  * Generates a data received message at the given dest node.
271  *
272  * @param gid The GID of the destination node.
273  * @param lp The logical processor initiating the request.
274  * @param msg A message containing information about the data and it's source.
275  * @param ts How far in the future this event should fire.
276  *
277  * @return the created event.
278  */
279 tw_event *genDataReceivedMessage(double gid, tw_lp *lp, sensor_message *msg,
280                                  tw_stime ts) {
281   tw_event *e;
282   sensor_message *m;
283
284   e = tw_event_new(gid, ts, lp);
285   m = tw_event_data(e);
286   copyMessageData(msg, m);
287
288   m->type = DATA_RECEIVED;
289   m->last_last_gid = msg->last_gid;
290   m->last_gid = lp->gid;
291   return e;
292 }
293
294 /**
295  * Sends a message originating from s to all nearby sensors.
296  *
297  * @param s The originating sensor
298  * @param lp the associated lp.
```

```
299   * @param type The event type to send
300   * @param msg A message that is associated with the generated message.
301   */
302  void sendMessage(sensor_state *s, tw_lp *lp, sensor_event_t type,
303                   sensor_message *msg) {
304    //sensor_message *m;
305    tw_event *e = NULL;
306
307    // Take care of events that alter the state of the originating sensor
308    switch (type) {
309      case SINK_DISCOVERED:
310        add_sink(s, lp->gid, lp->gid, 0, s->power);
311        type = SINK_LOC;
312        break;
313      default:
314        break;
315    }
316
317    // Send the requested message if valid type to all nearby neighbors
318    int i = 0;
319    tw_stime ts;
320    if (type == DATA_RECEIVED) {
321      INIT_TS_TRANSMIT(msg->data_size / gtransmit_rate);
322    } else {
323       INIT_TS_TRANSMIT(transmission_delay);
324    }
325    // Ensure it's positive
326    if (ts < 0) {
327      ts *= -1;
328    }
329    s->total_transmit_time += ts;
330
331    s->power -= power_loss(ts, transmit_power);
332
333    for (i = 0; i < s->num_nearby; ++i) {
334      switch (type) {
335        case SINK_LOC:
336          e = genSinkLocMessage(s, s->nearby_Sensors[i], lp, ts);
337          break;
338        case SINK_LOST:
339          e = genSinkLostMessage(s->nearby_Sensors[i], lp, ts);
340          break;
341        case DATA_RECEIVED:
342          e = genDataReceivedMessage(s->nearby_Sensors[i], lp, msg, ts);
343          break;
344        default:
345          // These types should not be associated with this type
346          // of message sending.
347          break;
348      }
349      if (e != NULL) {
350        tw_event_send(e);
```

```
351         printMessageInfo(lp->gid, s->nearby_Sensors[i], tw_now(lp));
352         // The multiple messages are used for the simulation. In reality
353         // this would be done by a single transmission.
354       } else {
355         printf("WARNING: Event Not Sent. Type: %d\n", type);
356       }
357     }
358     if (e != NULL) {
359       s->messages_sent++;
360     }
361   }
362
363   /**
364    * Starts the process of routing data to a sink.
365    * Sensors self select according to a backoff timer.
366    * This function send the initial DATA_RECEIVEd to all nearby sensors
367    * and waits for a response with the same nonce.
368    *
369    * @param s The sensor state.
370    * @param lp The processor associated with the lp
371    * @param msg If null, indicates that this is the start of sending data.
372    *            If not null, we are forwarding received data.
373    */
374   void send_data(sensor_state *s, tw_lp *lp, sensor_message *msg) {
375
376     // If we have knowledge of a sink AND
377     // If we have data
378     if (s->trying_to_send == 0 && s->sink_gid[0] != UNKNOWN &&
379         s->data_size > 0) {
380       // Send data to all neighbors
381       double data_size = s->data_size;
382       int wasCreated = 0;
383       if (msg == NULL) {
384         msg = (sensor_message *)malloc(sizeof(sensor_message));
385
386         msg->source_gid = lp->gid;
387         msg->dest_gid = s->sink_path[0][0];
388         msg->nonce = s->messages_sent;
389         msg->data_size = data_size;
390         msg->start_time = tw_now(lp);
391         msg->num_nearby = s->num_nearby;
392         msg->last_gid = lp->gid;
393         msg->last_power = s->power;
394         wasCreated = 1;
395       } else if (msg->source_gid == lp->gid) {
396         // If this is the original source, combine all data possible
397         msg->data_size = data_size;
398       }
399
400       // Store this so data can be combined.
401       s->trying_to_send = 1;
402
```

```
403      // Update the stats for this sensor
404      msg->last_power = s->power;
405      msg->num_hops = s->sink_dist[0][0];
406      msg->sink_gid = s->sink_gid[0];
407
408      // Send the data to all neighbors
409      sendMessage(s, lp, DATA_RECEIVED, msg);
410
411      // Start the wait timer
412      sendSelfMessage(s, lp, WAIT, msg);
413
414      if (wasCreated == 1) {
415        // free up created memory
416        free(msg);
417        msg = NULL;
418      }
419    }
420  }
421
422  /**
423   * After receiving instruction to forward the data, ACKS to the previous
424   * sensor and then forwards the data.
425   *
426   * @param s The sensor that received data.
427   * @param lp The logical processor associated with teh sensor.
428   * @param msg The message containing information about the data to forward.
429   */
430  void forward_data(sensor_state *s, tw_lp *lp, sensor_message *msg) {
431    // In the real world, this ack would be combined with the data
432    // forward, ergo no extra transmission time.
433    tw_event *e = genAckEvent(lp, msg);
434    tw_event_send(e);
435
436    printMessageInfo(lp->gid, msg->last_last_gid, tw_now(lp));
437
438    // If this is the sink, stop the message from forwarding
439    if (s->sink_dist[0][0] == 0) {
440      // Don't increase the data size, assume it goes away immediately.
441      // update stats
442      transmission_recv_ct++;
443      transmission_path_time_total += tw_now(lp) - msg->start_time;
444      data_sink_received += msg->data_size;
445    } else {
446      msg->last_last_gid = msg->last_gid;
447      msg->last_gid = lp->gid;
448      msg->dest_gid = s->sink_path[0][0];
449
450      send_data(s, lp, msg);
451    }
452
453  }
454  #endif
```

Listing A.13: Source Code

```
1  /**
2   * @file assisted/wsn.h
3   * @brief Defines the structures and any KASS specific variables.
4   * @author Thomas Reale
5   * @date 12-2011
6   * @version 1.0
7   *
8   * Defines the message structure, the sensor structure, and possible events.
9   *
10  */
11
12 #ifndef INC_wsn_h
13 #define INC_wsn_h
14
15 #include <ross.h>
16
17 /** typedef to identify an event */
18 typedef enum sensor_event_t sensor_event_t;
19 /** typedef to define a sensor's state */
20 typedef struct sensor_state sensor_state;
21 /** typedef to define a message between sensors */
22 typedef struct sensor_message sensor_message;
23 /** typedef to define a location */
24 typedef struct Location Location;
25 /** typedef to define a sensor structure */
26 typedef struct Sensor Sensor;
27 /** typedef to define event data */
28 typedef struct Event Event;
29
30 /** @def MAX_SINKS
31  * Used to set the maximum number of sinks stored */
32 #define MAX_SINKS 5
33
34 /** @def MAX_PATHS_PER_SINK
35  * Defines the maximum number of alternate paths to each sink */
36 #define MAX_PATHS_PER_SINK 3
37
38 /** @def MAX_WAIT
39  * used to set the maximum number of events to wait on */
40 #define MAX_WAIT 10
41
42 /** Keeps track of all sensors to allow to simulate distance between sensors */
43 static Sensor **sensors_list;
44
45 /** The possible events associated with a sensor */
46 enum sensor_event_t
47 {
48   SLEEP = 1,
49   AWAKE,
50   DEAD,
51   INIT,
```

```
52     SINK_LOC,
53     SINK_LOST,
54     SINK_DISCOVERED,
55     DATA_GEN,
56     DATA_RECEIVED,
57     DATA_EXPIRED,
58     DATA_FORWARD,
59     DATA_ACK,
60     WAIT
61   };
62
63   /** contains a cartesian coordinate */
64   struct Location {
65     /** The horizontal position of a sensor */
66     long x;
67     /** The vertical position of a sensor */
68     long y;
69   };
70
71   /** Information about a sensor for others to use for communication */
72   struct Sensor {
73     /** The location of this sensor */
74     Location location;
75     /** The global ID of this sensor */
76     long gid;
77   };
78   /**
79    * Contains the information related to a single sensor
80    * including statistics.
81    */
82   struct sensor_state
83   {
84     /** The state of the sensor (AWAKE/SLEEP/TRANSMIT) */
85     int state;
86     /** Indicates that this sensor is trying to send data */
87     int trying_to_send;
88     /** The power remaining on the sensor in mAH */
89     double power;
90     /** Location of the sensor */
91     Location location;
92     /** The data size on the device in bytes */
93     double data_size;
94     /** The GID to communicate with in order to send data to a sink.
95      * corresponds with sink_gid. Paths are ordered according to dist
96      */
97     double sink_path[MAX_SINKS][MAX_PATHS_PER_SINK];
98     /** The priority of each path */
99     double sink_cost[MAX_SINKS][MAX_PATHS_PER_SINK];
100
101    /** The distance to the data sink. 0 indicates either that
102     *  this is the sink a negative value indicates no sink.
103     *  corresponds with sink_gid. This value is calculated as a priority
```

```
104      *  based on hop count and power
105      */
106     double sink_dist[MAX_SINKS][MAX_PATHS_PER_SINK];
107     /** The ID of the sink, to stop continuous broadcasts. Array is ordered
108      * as inserting and deleting is less frequent than accessing */
109     double sink_gid[MAX_SINKS];
110     /** A list of the GIDs for nearby sensors. */
111     long *nearby_Sensors;
112     /** The number of nearby sensors */
113     int num_nearby;
114     /** Events that may need to be canceled if another event occurs */
115     sensor_message *wait_events[MAX_WAIT];
116     /** the number of wait events in the queue. */
117     int num_wait_events;
118
119     /** An indicator that says the sensor should go to sleep when possible */
120     int wants_to_sleep;
121     /** The time that this sensor last changed states */
122     tw_stime last_state_change;
123
124     // Used for statistics
125     /** Total number of times awake */
126     double times_awake;
127     /** Total number of times asleep */
128     double times_asleep;
129     /** Total number of messages sent */
130     double messages_sent;
131     /** Total awake time */
132     tw_stime total_awake_time;
133     /** Total asleep time */
134     tw_stime total_asleep_time;
135     /** Total transmit time */
136     tw_stime total_transmit_time;
137   };
138
139   /**
140    * A message to trigger events
141    */
142   struct sensor_message
143   {
144     /** The type of event this message is associated with */
145     sensor_event_t type;
146     /** The data size contained in this message (DATA RECEIVED/DATA GEN) */
147     double data_size;
148     /** The source gid of the original message */
149     double source_gid;
150     /** The destination gid of this message */
151     double dest_gid;
152     /** The number of neighbors near the most recent sender */
153     double num_nearby;
154     /** The sensor who was previous to the last sensor who forwarded the data.
155      * This is used to update a sensor who receieved back a forwarded message.
```

```
156      * It is much simpler than keeping track of it in the sensor */
157     double last_last_gid;
158     /** The power the last sensor this message crossed had */
159     double last_power;
160
161     // Fields for messages originating with sinks In SINK_LOC message,
162     /** Number of hops to sink. When transmitting data contains expected
163      * number of hops to sink */
164     int num_hops;
165     /** In SINK_LOC message, gid of last location on path to sink. */
166     double last_gid;
167     /** The gid of the sink */
168     double sink_gid;
169     /** An id unique to the origin sensor */
170     int nonce;
171     /** The time that this message was initially sent, used for statistics */
172     tw_stime start_time;
173     /** The time that this specific message was send from the previous sensor */
174     tw_stime time_sent;
175 };
176
177 #endif
```