EMPIRICAL ANALYSIS AND OBSERVATIONS OF ROUTING PROTOCOLS FOR WIRELESS SENSOR NETWORKS

BRETT M. PARSONS

Empirical Analysis and Observations of Routing Protocols for Wireless Sensor Networks

by

Brett M. Parsons

A thesis submitted to the

School of Graduate Studies

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

Memorial University of Newfoundland

November 2006

St. John's Newfoundland

Your file Votre référence
ISBN: 978-0-494-31274-2
Our file Notre référence
ISBN: 978-0-494-31274-2

Abstract

Routing in Wireless Sensor Networks poses unique challenges due to the low-energy requirement and the resource-constrained nature of nodes in the network. This brings about a need for efficient routing protocols for Wireless Sensor Networks. We selected three routing protocols for Wireless Sensor Networks (Directed Diffusion, Dynamic Source Routing, and Minimum Transmission Routing) and implemented them in a novel software framework. A simulation study was carried out to evaluate these routing protocols in regards to typical Wireless Sensor Network deployment scenarios, with a focus on the application of habitat monitoring. Additionally, well-defined metrics were used to measure the performance of each routing protocol in the experimentations. Based on the results of our experimentations, we recommend Dynamic Source Routing as the preferred routing protocol in most Wireless Sensor Network deployments for habitat monitoring purposes.

Acknowledgements

Foremost, I would like to thank my supervisor, Dr. Rodrigue Byrne, for the guidance and knowledge imparted during the course of this thesis. The patience and dedication you displayed are unmatched and greatly appreciated. Additionally, I would like to thank my family and friends who supported me during the entire process. Without your encouragement, this thesis would not have been possible. Finally, I would like to thank all the members of the TinyOS mailing list. Your willingness to lend a hand when needed greatly contributed to the completion of this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1 – Introduction

1.1 Overview

Habitat monitoring is a prominent application for sensor networks and serves a very positive purpose. This application involves using a sensor network to monitor various phenomena such as temperature, light, or humidity in an effort to better understand a particular habitat (e.g., a nesting ground for birds). Additionally, routing protocols play an important role in such applications due to the short communications radius of nodes in the network. As a result, this thesis focuses on the problem of determining the most efficient routing protocol for sensor networks that are deployed in habitat monitoring scenarios.

There is currently a fair amount of research on new routing protocols for sensor networks. Many interesting protocols have been proposed and are currently being used in production sensor networks. This means that researchers interested in deploying a sensor network have many options in regards to what routing protocol to use. Unfortunately, there is no substantial research on which routing protocols are most efficient in particular sensor network applications (e.g., habitat monitoring). As more researchers begin to utilize sensor networks to assist in their experiments, it is important to supply these people with enough relevant information so they can select the routing protocol best suited to their needs.

This thesis evaluates three routing protocols for sensor networks: Directed Diffusion, Dynamic Source Routing, and Minimum Transmission Routing. The evaluation is done by implementing the protocols in a common framework and performing simulation studies to determine how the protocols perform in relation to well-

defined metrics. TinyOS is used as the operating system for the sensor nodes while the TOSSIM simulator is utilized to facilitate the experimentations.

The metrics used to evaluate the routing protocols are chosen for their relevance to habitat monitoring applications. To evaluate the energy efficiency of the routing protocols, a custom metric called Average Dissipated Energy is used. This metric measures the average amount of energy required for a single data message to be routed from a source node to the base station. Other energy-related metrics include Energy Consumption Variation (the variance of energy dissipation amongst all nodes in the network) and Total Number of Messages Transmitted (the number of messages required for correct operation of the routing protocol). In addition, an Average Delay metric is used to measure the average time taken to route messages to the base station and a Code Size metric is utilized to determine the amount of resources (i.e., memory) required for the correct operation of each routing protocol.

Additionally, several other metrics are used to evaluate how well routing protocols recover from the occurrence of a network error. For example, Recovery Total Energy Consumption measures how much energy is required to stabilize the network after an error occurs. Also, the Stabilization Time metric is used to determine how long it takes the network to recover from an error. Finally, the Number of Data Messages Lost metric indicates how many data messages are lost in the event of a network error.

1.2 What are Sensor Networks?

Sensor networks have become a hot topic in networking research in recent years and many people are excited about the advancements being made in the field. This emerging

wireless technology could greatly impact the world that we live in, and has almost limitless applications.

Sensor networks consist of many tiny devices, called sensor nodes (or motes), which are capable of detecting physical stimuli such as light, temperature, and sound. Besides sensory components, sensors contain a microcontroller, memory, and a wireless transmitter/receiver [1]. A group of sensor nodes in a given area can form a network amongst themselves (with the help of routing and self-formation algorithms). Sensor nodes can either perform computations on data locally or transmit their sensor readings back to a base station. The base station is often a computationally rich device such as a laptop and is responsible for collecting and performing computations on sensor data.

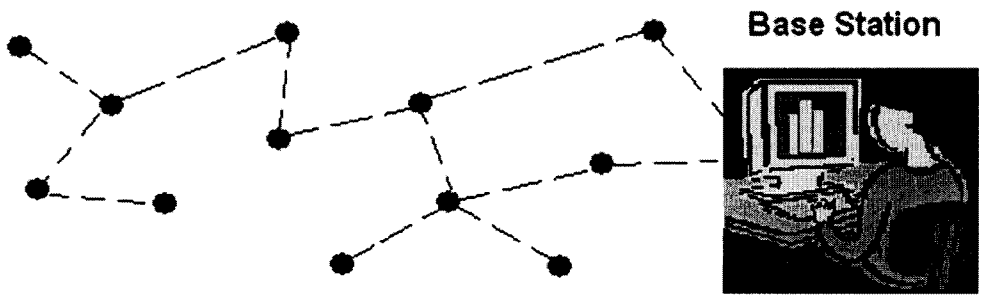


Figure 1 - A sensor network with a base station

Figure 1 shows a typical sensor network communicating with a base station. Here, the dotted lines represent wireless links between sensor nodes (represented by the dots).

1.3 How do Sensor Networks Differ From Other Wireless Networks

Although sensor networks may appear to be similar to many other forms of ad hoc networks (such as 802.11), there are some subtle differences.

First of all, sensor networks don't require any fixed infrastructure (except perhaps for a base station) [2]. They are purely ad hoc in nature meaning that nodes may join (i.e., activate) or leave (i.e., fail) constantly.

Secondly, individual sensor nodes are designed to be inexpensive as they may be purchased in large quantities for many purposes [1]. As well, sensor nodes may be disposable because they can be deployed in locations where they are not easily retrievable (e.g., a remote island). The low cost requirement means that sensor nodes are severely resource constrained. For instance, the typical sensor node contains only a 4 MHz microprocessor [3]. Also, the transmission range of the nodes is very small, measuring only several meters depending on the deployment environment [4]. Memory resources are also scarce as most sensor nodes have only 128 K of program memory and 512 bytes of data memory.

Finally, since sensors run on battery power, energy consumption is a huge concern [2]. In many cases, there is no way to replace a sensor's battery, resulting in a node failure once the battery is depleted. This brings about a need for energy efficient algorithms for sensor networks.

1.4 Applications of Sensor Networks

Although there are countless applications of sensor networks, only a few of them are examined in this section.

Currently, one of the main applications of sensor networks is for habitat monitoring; this is also the application focused on in this thesis. Sensor networks allow us to collect data in complex ecosystems, such as collecting temperature readings or

monitoring different substances found in the soil and the air [5]. This data can then be used to better understand the ecosystem and perhaps develop a solution to conserve it.

Sensor networks offer many advantages over the traditional methods of collecting environmental data. Instead of sending teams into the field to manually observe and record results, researchers can set up a sensor network to measure the desired phenomena and access the results remotely [3]. Also, some sites might be difficult for humans to access on a continual basis (e.g., isolated islands) and are ideal for monitoring via a sensor network. Additionally, it may be difficult for a researcher to get close enough to the specimen being studied without being intrusive. For example, some animals may alter their behavior in the presence of humans. Sensor nodes, however, are unobtrusive and allow the animals to be monitored without any interference.

One example of the use of sensor networks for habitat monitoring comes from the PODS project at the University of Hawaii. This project is using sensor networks in an effort to remotely monitor rare and endangered plant species on the island of Hawaii [6]. By measuring rainfall, wind, temperature, humidity, and solar radiation, the researchers have access to information that allows them to make the best decisions possible to protect these endangered plant species.

Another example of sensor networks being utilized for habitat monitoring is the Great Duck Island project by the Center for Information Technology Research in the Interest of Society (CITRIS) [3]. In this case, a group of CITRUS researchers from the University of California at Berkeley have teamed up with conservation biologists from the College of the Atlantic to monitor the breeding habits of seabirds off the coast of Maine. Researchers started off by placing sensor nodes in burrows inhabited by the

seabirds. Biologists can then remotely monitor when a burrow is occupied by a bird using an infrared heat sensor built into the sensors nodes. Before the sensor network was setup on the island, the only way for biologists to observe the behavior of the seabirds was through carefully planned trips using a portable video system and human observations. However, since Great Duck Island is remote and difficult to access, the sensor network significantly improves the way in which biologists can gather the information they need.

The original Great Duck Island sensor network deployment in 2002 contained 32 UC Berkeley motes using TinyOS as their operating system [3]. These motes contain an Atmel microcontroller with a clock speed of 4 MHz and 512K of non-volatile storage. The on-board radio in these motes has a bidirectional data rate of 40 kbps. As well, each mote was powered by two AA batteries with an expected lifespan of approximately 6 months. Routing in the sensor network was handled using a simple, hierarchical approach; a formation algorithm was run on the network to create a routing tree with nodes on different levels depending on their distance from the base station.

The uses of sensor networks are not limited to habitat monitoring, however. On the other side of the spectrum, sensor networks can have a military purpose.

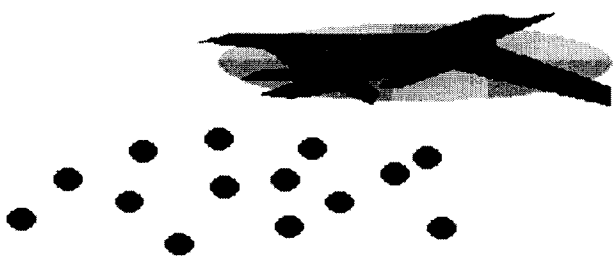


Figure 2 - Dispersing sensors from a plane

As seen in [7], sensor networks can be used to monitor hostile battlefields without endangering the lives of soldiers. In these experiments, unmanned aerial vehicles (UAV's) were used to fly over a mock battlefield and drop sensor nodes (as seen in Figure 2). Upon landing, the sensor nodes formed a network and began collecting readings. These readings can aid in determining enemy troop movements without any assistance from human personnel. In order to receive the sensor readings, however, the UAV has to fly over the battlefield within transmission range of the sensor network. It may also be necessary to add security measures to the nodes to prevent adversaries from eavesdropping or even tampering with the data being transmitted [8].

Another use for sensor networks involves object tracking. As an object travels through a sensor network, different nodes sense it and relay this data back to the base station. An example of an object-tracking application is presented in [9], where the authors deploy a sensor network for the purposes of tracking and intercepting vehicles moving through a sensor network. Additionally, in [4], the authors present the Frisbee model as a means for minimizing energy consumption among nodes while tracking objects in the sensor network.

The final example of a sensor network application is measuring the structural integrity of buildings [10]. Sensor nodes are placed on key pillars or other supports to monitor their integrity (i.e., – how much force is being exerted). If there are any signs of danger, the base station can be immediately alerted so that the proper security measures are taken.

As is evident in the above examples, the practical uses of sensor networks are quite varied and are constantly expanding.

1.5 Overview of routing in sensor networks

1.5.1 The Need for Routing in Sensor Networks

A typical sensor network may be comprised of many nodes. In fact, depending on the phenomena being sensed, it is possible for the network to consist of hundreds or thousands of nodes. Due to the limited transmission range of the radio contained in a sensor node, a node is only able to communicate directly with a small subset of the other nodes in the network. Take, for example, the sample sensor network shown in Figure 3.

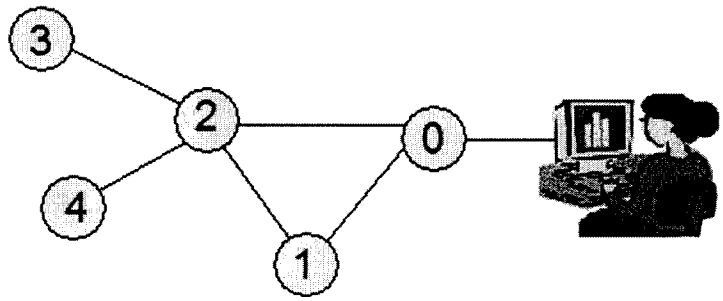


Figure 3 - A sample sensor network

Here, the circles represent sensor nodes in the network, the lines determine which nodes are in transmission range of each other, and the computer represents the base station. For instance, node 2 can hear radio transmissions from node 3, but node 0 cannot. This simple example displays the need for routing algorithms in sensor networks. In order for node 3 to send a message to the base station, the message needs to be routed through nodes 2 and 0. Vice versa, if the base station wanted to send a message to node 3 (e.g., requesting the node to start recording data), the transmission path would be reversed.

1.5.2 Challenges of Routing in Sensor Networks

Sensor networks have some unique properties that can complicate the process of routing messages from a source to a destination.

The first such property is the lack of a fixed network infrastructure [2]. Sensor networks tend to be very dynamic with nodes continually failing/rejoining the network. This means the network topology can change very rapidly; a property that most conventional routing protocols are not designed around. Also, unlike traditional networks, there are no specialized devices such as routers to help forward messages to their destination. In fact, every node in a sensor network must act as a router in order for messages to be received correctly. Without this assumption, the network would not function properly.

Another unique property that poses a challenge for routing in sensor networks is that nodes are powered by a battery [2]. As such, when the battery in a node fails, the node itself will fail. If the same route is used continuously to send messages from a source to a destination, the nodes on this route will be sending more messages per capita than other nodes in the network. Since radio transmission is the main source of battery usage in sensor nodes, the nodes on the chosen route will deplete their batteries at a faster rate, thus failing more quickly than the other nodes. To see an example of this, refer to Figure 4.

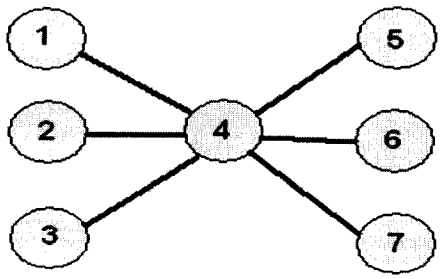


Figure 4 - Another sample topology

In this simple network topology, node 4 is acting as a bridge between two sets of nodes. Thus, whenever a message needs to be sent from a node on the left side to a node of the

right side, it must pass through node 4. As a result, node 4 consumes energy at a faster rate than other nodes in the network, leading to a shorter lifespan. To combat this problem, routing protocols must be mindful of continually choosing the same path from source to destination in an effort to evenly distribute the energy consumption.

Perhaps the most important sensor network property that creates obstacles in routing is the constrained resources available to the nodes in the network. As mentioned above, each sensor node contains only a modest processor (e.g., 4 MHz) and a small amount of memory (e.g., 512 bytes of data memory) [3]. This implies that only a small amount of data can be stored at any time. In terms of routing, it follows that computationally expensive protocols are not suitable for sensor networks. As well, protocols that require a large routing table in order to properly send messages are not appropriate.

As illustrated above, the majority of research completed on routing protocols for ad-hoc networks is not directly applicable to sensor networks. Most ad-hoc routing protocols are designed with devices such as laptops, cell phones, and PDAs as likely members of the network; however, even these devices are computationally rich compared to sensor nodes. Therefore, popular ad-hoc routing protocols such as Ad-hoc On-demand Distance Vector (AODV) [11] and Destination-Sequenced Distance Vector (DSDV) [12] are not suitable for sensor networks [13], due to their requirements of storing potentially large routing tables and a necessity for periodic routing update messages. This also brings about a need for routing protocols designed specifically for sensor networks [14].

1.6 Research Approach

In order to accomplish the tasks outlined above, several of the most widely used sensor network routing protocols for the application of habitat monitoring are selected: Directed Diffusion, Dynamic Source Routing, and Minimum Transmission Routing. Each of these protocols provides a unique approach to routing in sensor networks.

After selecting the routing protocols to examine, a set of metrics is chosen that are used to compare the performance of each routing protocol. Among the metrics selected are energy efficiency and code size.

The next step in the process is selecting the tools needed to carry out our analysis. TinyOS is chosen as the operating system for the sensor nodes. TinyOS is an open-source operating system specifically designed for sensor networks and is widely used in the sensor network community [15].

As well, since the analysis requires a large number of sensor nodes, a simulator is used to carry out the experimentations (as opposed to using actual hardware). For this purpose, TOSSIM, a discrete-event simulator for TinyOS networks [16], is selected. Another simulation package, the AVR Simulation and Analysis Framework (AVRORA) [17], was also evaluated. However, due to factors including poor support for custom radio models as well as a lack of community involvement, this package was ultimately abandoned in favor of TOSSIM.

At this stage, an experimentation plan is compiled to determine what experiments are necessary to accurately compare the routing protocols against the selected metrics. The experiments are then run and the results documented. The experimentation plan is detailed in Chapter 3.

To perform the experiments, each routing protocol is implemented for execution on the TinyOS operating system. This also involves devising custom software architecture for routing in sensor networks due to a lack of current approaches. This architecture is presented in Chapter 4.

With the data collected from the experiments, conclusions are drawn as to what routing protocols perform best with respect to each metric. The experimentation results are found in Chapter 5 while the conclusions are offered in Chapter 6.

1.7 Assumptions/Assertions

In choosing which routing protocols to examine, it is important to outline the assumptions made in regards to the operation of the sensor network.

The first assumption is that the sensor network is interest-based, or data-centric; a concept introduced in the Directed Diffusion paradigm [18]. An interest-based network entails that the base station broadcasts interest messages when it needs to request data (described in more detail in Chapter 2). These interest messages specify the type of data to be collected, the frequency and duration it should be collected for, and the criteria that the data should match. For example, an interest message can specify that nodes send back temperature readings greater than 30 degrees Celsius.

Node ID:	12
Reading Type:	Temperature
Reading:	46 degrees Celsius
Timestamp:	12:25:45
Interest ID:	53

Table 1 - Sample data reading

Table 1 shows an example of a data message that might be sent by a node in response to this interest message. This data message explains that a node with ID 12 recorded a temperature reading of 46 degrees Celsius at time 12:25:45. Additionally, the Interest ID field states that this data is associated with interest 53.

The use of criteria to specify which nodes should send back data is quite different from traditional address-centric routing. In address-centric routing, messages are routed from a source to a particular destination (e.g., The base station requests data from node X). The interest-based routing demonstrated above, however, is a form of data-centric routing [18]. In data-centric routing, the base station sends interests for named data as opposed to sending a request to a particular node. This allows users to make more generalized queries to the network such as "What nodes are registering temperature readings above X degrees Celsius?" or "Which burrows are currently inhabited by a bird?".

It is important to note, however, that interest-based routing does allow requesting data from one particular node. There may be many instances where a researcher is only concerned with sensor readings at a particular point in the network. In this case, the interest criteria would be that only specific nodes send data back to the base station (e.g., Give me all sensor readings being recorded by sensor node X). The use of criteria allows the routing protocol to scale depending on the use of the sensor network. In practice, every sensor network may have a different set of criteria for determining how interests are matched to sensor data.

The second assumption made is that nodes are not able to determine their geographic position. There are several routing protocols for sensor networks, such as

[19], that assume the sensor nodes are equipped with a Global Positioning System (GPS). The approach these protocols take is to route messages in the general direction of the base station. They argue that most sensor networks will be GPS-equipped because it is important to know the locations of nodes that are sending back data to the base station. Although this statement is logical, including GPS technology in the sensor nodes can significantly increase the costs of deploying the network. By keeping deployment costs lower, more researchers are able to utilize sensor network technology in their experiments.

There are methods that allow all nodes in a sensor network to determine their geographic position while only having the base station or possibly a small subset of sensor nodes equipped with a GPS. In [20], researchers use the base station to emit several beams of laser light on different axes. These beams can be detected by each node in the sensor network and used to calculate their position in three dimensions relative to the base station. However, this is not suitable for all habitat monitoring purposes because it assumes that every node is in line of sight with the base station. For example, if sensor nodes were placed in underground bird burrows, this method would not be feasible. Additionally, in [21], researchers use a small number of beacon nodes (i.e., nodes that are aware of their geographic position) to continually broadcast their position. Nodes that receive this broadcast then use the received signal strength to make estimations of their own position. This approach may not be suitable for habitat monitoring applications, however, because received signal strength can fluctuate heavily depending on the environment that the sensor nodes are deployed in. As well, obstacles in the sensor

network can interfere with received signal strength (e.g., a deer walking between two sensor nodes).

The final assumption made regarding the operation of the sensor network is that the interest messages are propagated throughout the network via flooding. Each node that receives an interest will re-broadcast it to all neighbours. Although this is inefficient, it is the only viable approach when nodes have no knowledge about the overall network topology [18].

1.8 Outline

In Chapter 2, a detailed analysis of the Directed Diffusion, Dynamic Source Routing, and Minimum Transmission Routing protocols is provided. The algorithms used by all three protocols are explained and, in addition, the advantages and disadvantages of each protocol are presented.

Chapter 3 details the experimentation plan as well as the motivation for performing the experiments. As well, the metrics used in the experimentations are explained. Additionally, the primary tools used to carry out the experiments are described with the reasons for selecting each of them.

In Chapter 4, a novel software architecture for implementing routing protocols in interest-based sensor networks is presented. Also, the experimentation setup is detailed to explain how the chosen software packages collaborate to carry out the experiments.

Chapter 5 presents the results of the experimentation plan. The outcomes for all three routing protocols are presented on a metric-by-metric basis. Additionally, an analysis is performed to justify the results of each experiment.

Finally, Chapter 6 includes recommendations as to which routing protocol is better suited for habitat monitoring applications. As well, a direction for future work is proposed to expand on the research conducted in this thesis.

Chapter 2 – Literature Survey

2.1 Overview

In this chapter, the three routing protocols selected for the analysis are discussed: Directed Diffusion, Dynamic Source Routing, and Minimum Transmission routing. The advantages and disadvantages of each protocol are detailed as well. Also discussed are the tools used in carrying out the experimentations, which include TinyOS, TOSSIM, PowerTOSSIM, and Tython.

2.2 Directed Diffusion

Directed Diffusion is an important contribution to sensor network routing because it introduces the concept of data-centric routing to sensor networks [18].

Using the Directed Diffusion paradigm, data generated by sensor nodes are assigned an attribute-value pair [18]. For example, an attribute might be temperature and the value could be 15 degrees Celsius. In data-centric routing, requests are then made for named data; in other words, all data matching a set of attribute-value criteria (e.g., all temperature readings greater than 15 degrees Celsius). This differs from the traditional address-centric routing approach where a request is made for data from a particular node (i.e., send this message to the node with address X).

2.2.1 Interest Dissemination

In Directed Diffusion, when the base station wishes to obtain data from the network, an interest message (sometimes referred to as a sensing task) is created. The interest

message contains a list of the aforementioned attribute-value pairs in order to specify what data should be collected. An example of an interest message can be seen in Table 2.

Id	53
Type	Temperature
Criteria	> 40 Celsius
Interval	30 ms
Duration	60 s

Table 2 - Sample Interest Message

Let's assume the base station broadcasts the above interest message into the sensor network. The interest message requests that sensor nodes send temperature readings matching a certain criteria. Here, the criterion is that temperature sensor readings are above 40 degrees Celsius. Therefore, nodes will only send back data if this condition is met (i.e., temperature readings that are less than 40 degrees Celsius are not transmitted). Finally, the interest message specifies that, if the above criterion is met, readings should be sent at an interval of 30 milliseconds for a duration of 60 seconds.

Each node in the network that receives an interest message stores it in a cache and rebroadcasts the message. In this way, the interest message can reach all nodes in the network through flooding. However, if the node has already seen this interest message (i.e., the interest is in the cache of recently seen interest messages), the message is quietly discarded. This allows the interest message to be damped and prevents nodes from continuously broadcasting identical interest messages.

The initial interest message sent by the base station has a low interval defined and is considered an exploratory interest. This results in nodes sending data back to the base

station at a low rate. The reasoning for this approach is that the base station can get a good idea of what neighbours are capable of providing the most appropriate data matching the interest (this is application-specific). Once the base station has this initial information, it can ask a preferred neighbour to send data at a higher interval.

Consider, for example, a radiation sensing network that measures the level of radioactivity to determine if it would be hazardous to the health of humans. Such an application is detailed in [22]. In this type of network, an interest can be sent asking for readings higher than a specific safety level. Although several neighbours may send data indicating higher than normal radiation readings, the user may be more interested in the neighbour reporting the highest readings. The selected neighbour can then be instructed to provide readings more frequently to help determine the rate at which the radiation is increasing. This technique is called interest reinforcement and is explained in section 2.2.3.

Another important aspect of interest propagation is that it makes use of localized interactions. This means that when a node receives an interest message, it considers the origin of that message to be the neighbour who sent it.

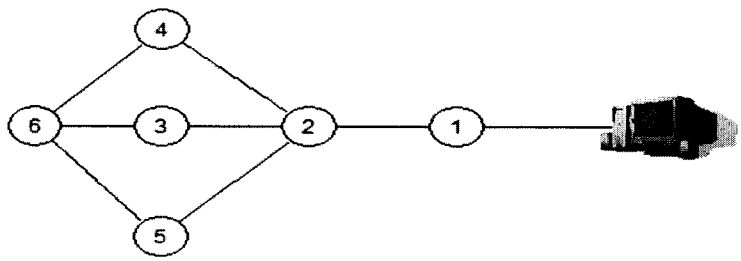


Figure 5 - Localized Interactions

In Figure 5, for example, assume that the base station (represented by the computer) broadcasts an interest message into the network. Now assume that this message arrives at node 3 after being rebroadcast by nodes 1 and 2. As far as node 3 is concerned, the

origin of the interest message is node 2. Therefore, when data that matches this interest is generated, the destination for the data will be node 2 and not the base station. This makes it possible for data aggregation to take place. For instance, if node 2 receives identical data from nodes 3, 4, and 5, it can consolidate these three messages into a single message and send the consolidated message back to the base station. This reduces data redundancy and improves energy efficiency since only one message needs to be sent as opposed to three.

It is also important to note that, due to dynamic network connectivity, interest messages may go unfulfilled. For instance, in Figure 5, if the base station sends an interest message to node 1 during a temporary loss of connectivity between the two nodes, the interest will never be received. To combat this, the base station can rebroadcast the interest message if it does not receive matching data after a specified period of time.

2.2.2 Data Delivery

Once a sensor node has cached an interest message, any data collected will be compared against the interest to determine a match. If the data matches any interests stored in the cache, the node must send the data to the neighbours who submitted a matching interest.

Each interest stored in the cache has a specified gradient [18]. A gradient is simply a direction in which to send the data (i.e., what neighbour sent me the interest?) and a data rate (i.e., at what rate should data be sent to the neighbouring node?).

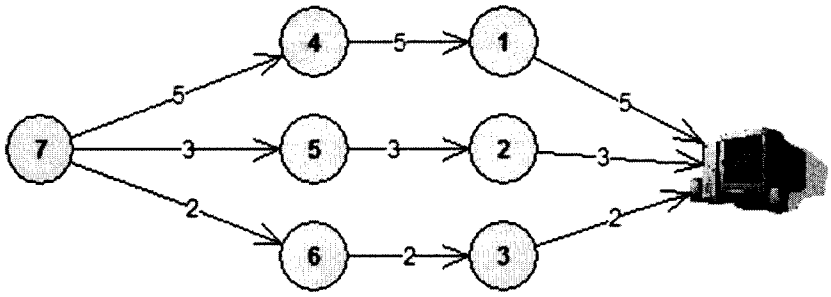


Figure 6 - Gradients

In Figure 6, the arrows between nodes represent gradients while the numbers on the arrows specify a data rate in events/sec. For example, the arrow between nodes 7 and 4 dictates that node 7 should send data messages to node 4 at a rate of 5 messages per second. Now assume that node 7 has collected data that matches interests sent from nodes 4, 5, and 6. Since each gradient has specified a different data rate, node 7 will select the *highest* rate among all gradients. Therefore, node 7 will send out data events at a rate of 5 per second.

Notice, however, that nodes 5 and 6 are now receiving data at a faster rate than specified by the gradients. In the Directed Diffusion paradigm, it is the responsibility of the receiving node to compensate for a faster than expected data rate. Thus, nodes 5 and 6 will have to normalize the data rate by perhaps dropping some of the received messages or by queuing the incoming messages. This may not be feasible, however, due to a limited amount of available memory.

When a node receives an incoming data message, it searches the interest cache to find any entries that match it. If no matching entries can be found, the received data message is dropped and will not be rebroadcast. If matching entries do exist, the data is transmitted to each applicable neighbour via the method described in the above paragraph.

Each node also maintains a data cache that contains recently seen data messages. When a data message arrives, it is compared to the data cache to ensure this message has not already been recently received. If the message does exist in the data cache, it is silently dropped and will not be processed. Both of these techniques provide protection against data redundancy and routing loops.

The above steps describe how data collected by the nodes is matched to interests and then transmitted through the network. By unicasting data along gradients, messages are pulled towards the base station along different paths, thereby satisfying the original interest message sent.

2.2.3 Path Reinforcement

Once the base station begins receiving data from its neighbours, it must choose a preferred neighbour for receiving data. For instance, the preferred neighbour could be the one providing the highest data rate, or perhaps the highest quality of data. In practice, the criteria used to select the preferred neighbour depend on the sensor network application. The act of selecting a preferred neighbour from which to receive data is called reinforcement. In the experiments, hop count is used as the criteria to select a preferred neighbour. That is, a node will reinforce the neighbour that minimizes the amount of hops data must travel to reach the base station.

To reinforce a particular neighbour, the base station creates an interest reinforcement message. This reinforcement message is identical to a regular interest message except that it specifies a higher frequency and duration. This instructs the receiving node to increase the interval at which it sends data to the base station as well as

the length of time data is collected. As well, unlike regular interest messages, the reinforcement messages are not flooded; instead, they are unicasted on a node-by-node basis until they reach the data sender.

Once a node receives an interest reinforcement message, it checks the cache to ensure the interest to reinforce actually exists. If it does not, the reinforcement message is silently dropped. If a matching interest is found, the receiving node updates its gradient with the new interval and duration values. Also, any node receiving an interest reinforcement message (with the exception of the base station) must also choose one of its neighbours to reinforce. The criteria for selecting this neighbour would presumably be the same as those used by the base station. By recursively selecting preferred neighbours, a reinforced path is created in the network through which data flows back to the base station.

An example of interest reinforcement can be seen in Figure 7. Here, assume that the base station has chosen node 3 as its preferred neighbour. The base station sends an interest reinforcement message to node 3 instructing it to send data messages at an interval of 5 ms for 120 s.

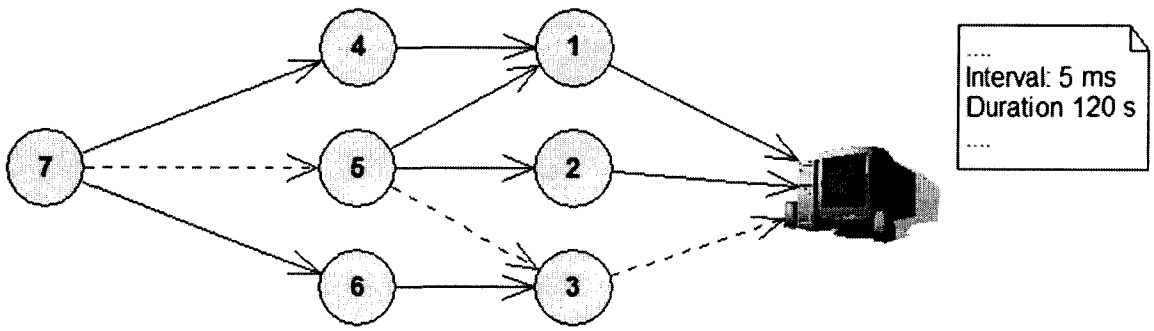


Figure 7 - Interest Reinforcement

Upon receiving the reinforcement message, node 3 must also select a preferred neighbour to reinforce. In this example, node 3 selects node 5 as the preferred neighbour and sends an interest reinforcement message to this node. Similarly, node 5 receives the message and chooses to send a reinforcement message to node 7 (which is actually the only choice). After this process, a reinforced path, denoted by the dashed arrows, exists from the base station to the source node (node 7).

Although Directed Diffusion ultimately uses a single reinforced path to route data back to the base station, there has also been research conducted to expand on this notion. For example, [23] presents a reinforcement technique that uses multiple braided paths for data routing to improve resilience against node failures.

Directed Diffusion also presents a concept called negative reinforcement [18]. Negative reinforcement involves tearing down unused data paths in the network. In Figure 7, for example, after the interest reinforcement has taken place, there are still paths that will continue to transmit data even though they were not reinforced (e.g., the path that includes nodes 7, 4, 1, and the base station). To negatively reinforce a path, there are several options available. One method is for the base station to send an explicit negative reinforcement message to all applicable neighbours. This message acts inversely to the reinforcement message but gets propagated through the network in a similar fashion. Perhaps the simplest method, however, is to let the original interests time out. For instance, if the original interest specified a duration of 60 seconds and was never reinforced, the interest would be dropped after this amount of time. Therefore, if a node receives a data message that matches the dropped interest, the data will be discarded.

2.2.4 Advantages and Disadvantages

The major advantage of Directed Diffusion is its energy efficiency when compared to flooding. Most of the energy savings are achieved by reinforcing only a small subset of nodes in the network to draw data back to the base station. Using this approach, the amount of messages transmitted is substantially better than a flooding approach, since messages are unicast back towards the base station.

Another advantage of the Directed Diffusion paradigm is the low overhead and complexity required to perform the algorithm. Since all message exchanges are localized, there are no expensive path computations required at the nodes. As well, no routing tables are required to be stored. In a network of thousands of nodes, storing routing tables requires significant amounts of memory. In Directed Diffusion, however, the only artifacts stored in memory are an interest cache and a list of recently seen data messages. This allows the protocol to scale well in large sensor networks since information is not required to be stored for each node in the network.

The adaptability of Directed Diffusion to dynamic network topologies is also a tremendous benefit. Sensor networks can have an extremely volatile topology with nodes failing and links degrading. The use of Directed Diffusion's reinforcement mechanism allows the protocol to recover quickly and efficiently in the case of most network failures. Once the base station detects that an error has occurred (i.e., it is no longer receiving data messages), it can reinforce another neighbour to continue receiving data.

Although Directed Diffusion has many advantageous properties, it also has some drawbacks. For instance, interest messages must be propagated through the network via flooding. This results in many messages being sent, increasing the energy consumption

of potentially all nodes in the network. This is not a huge concern if interest messages are only sent periodically. However, if interest messages are being consistently flooded, it will result in a faster depletion of energy at each node in the network.

Another issue with Directed Diffusion is that constantly using a single path for data delivery will lead to faster energy depletion for the nodes on this path. This situation might be avoided, however, by considering node energy levels when making reinforcement decisions on a local level. For instance, the base station would select the path containing nodes with higher energy levels to use when delivering data.

Additionally, the performance of Directed Diffusion can be negatively impacted in an error-prone network. If an error occurs after the network has stabilized (i.e., the base station is receiving data from a single neighbour), the base station may be forced to rebroadcast another interest message to rediscover new routes. If this occurs frequently, nodes will consume energy at a much faster rate, leading to a lower network lifetime.

2.3 Dynamic Source Routing

Dynamic Source Routing (DSR) is a well-known routing protocol used in multi-hop ad-hoc networks (MANETs). It is a lightweight, efficient protocol that was designed with resource-constrained devices in mind (e.g., cell phones, PDAs) [24].

The DSR protocol is comprised of two main parts; route discovery and route maintenance. Each node maintains a source route to any other node it wishes to communicate with. A source route is simply the sequence of hops necessary from the source node to reach the destination node. Each packet sent in the DSR protocol contains the required source route so that intermediate nodes can properly forward the

packet. If a node wishes to send to a destination for which it has no stored route, it must perform the route discovery process to find the route needed to transmit to the intended destination. This makes DSR a reactive routing protocol as routes are only acquired on an ad-hoc basis. Additionally, there are no periodic routing update messages of any kind.

Since it represents a class of existing multi-hop ad-hoc routing protocols, DSR provides a good benchmark with which to compare Directed Diffusion and Minimum Transmission Routing, which were developed specifically for sensor networks.

2.3.1 Route Discovery

When a node, Y, wishes to send a message to another node in the network, it checks a local route cache to determine if there is a stored route to the destination. If there is no route stored, the node must perform Route Discovery [24]. This process provides the source node with the sequence of hops necessary to reach the destination node. To initiate Route Discovery, the source node generates a Route Request (RRQ) message. This message contains the source node, destination node, a Route Request ID, and a route record. Combined with the node's unique address, the ID is a globally unique identifier for this RRQ and helps to prevent Route Request loops. The route record is populated as the RRQ propagates throughout the network because each node receiving the RRQ adds its identifier to the route record. The source node, Y, then broadcasts the RRQ message to its neighbours.

When a node, X, receives a RRQ, it checks to see if it is the destination of this RRQ. If node X is not the intended destination, it must ensure that this RRQ has not been seen before. To facilitate this, each node maintains a cache of recently seen RRQ IDs. If

node X determines it has already seen this RRQ, the message is silently dropped and not re-broadcast. Otherwise, node X will add its identifier to the route record and re-broadcast the packet to all neighboring nodes.

If node X is the intended destination, however, it must return a Route Reply (RRP) message to the source node [24]. The RRP message contains the accumulated record of hops in the route from source to destination. However, when sending the RRP back to the source node Y, it is important to note that node X cannot simply use the path accumulated in the route record of the RRQ since not all links in the network may be bi-directional. This can be caused, for example, if some nodes have a lower energy level, resulting in a shorter transmission radius. Due to this, if node X does not already have a cached route to the source of the RRQ (Node Y), it will have to perform Route Discovery of its own. In order to prevent an infinite Route Discovery loop, however, node X piggybacks the RRP on top of the RRQ.

Once the source node, Y, receives the RRP, it is stored in the route cache so that discovery does not need to be completed the next time data is to be sent. However, if the source node does not receive an RRP after a specified timeout value, it assumes the original RRQ was lost and generates a new one to broadcast to its neighbours. To avoid flooding the network with RRQs, an exponential back-off algorithm is used to determine when to send the next RRQ.

For an example of Route Discovery in DSR, refer to Figure 8. Here, assume that Node 4 in the sensor network wishes to send a data message back to the base station but does not have the required route in its route cache. To initiate Route Discovery, Node 4

creates a RRQ and broadcasts it. Eventually, the RRQ is received by the base station via the path 4-3-2-1.

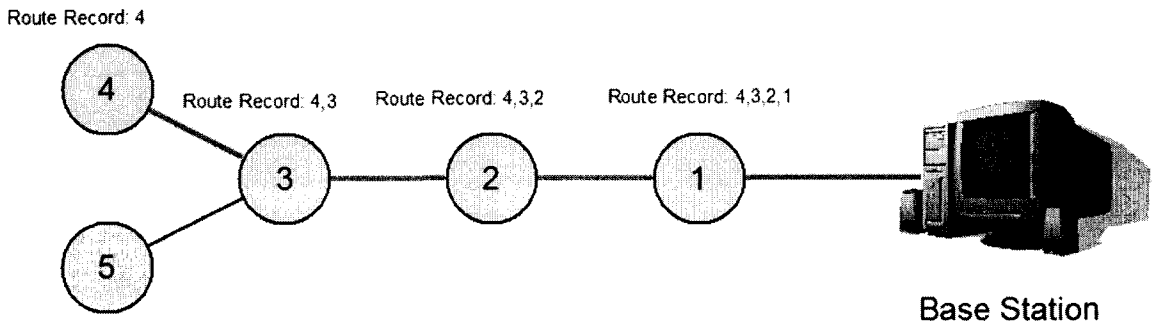


Figure 8 - Route Discovery

Once the base station receives the RRQ, it must generate an RRP. Assume that the base station has no stored route to Node 4. Since we are not assuming bi-directional links in the network, the base station will generate an RRQ of its own to find a route to Node 4 and piggyback the RRP on the RRQ.

2.3.2 Route Maintenance

As previously stated, DSR is designed around dynamic networks where the topology can frequently change. Therefore, stored routes that were previously valid may no longer be usable due to broken links or failed nodes. To recover from these types of failures, DSR incorporates route maintenance.

DSR operates on the principal that each node receiving a message is responsible for ensuring that the message successfully reaches the next hop on its way to the destination. The message is retransmitted to the next hop until confirmation is received (through an ACK mechanism) or the maximum number of retransmissions is reached.

Also, due to unidirectional links, the ACK message may have to travel across multiple hops to reach the sending node.

If a node is unable to send a message to the next hop, it must generate a Route Error (RERR) message to be propagated back to the node that originally sent the message. The Route Error message states that a problem has occurred in transmitting the message and specifies which link is broken. Once the source node receives the Route Error message, it removes the broken route from its cache. If the source has no other cached route to the destination, it must reinitiate the Route Discovery process to find a new, valid route.

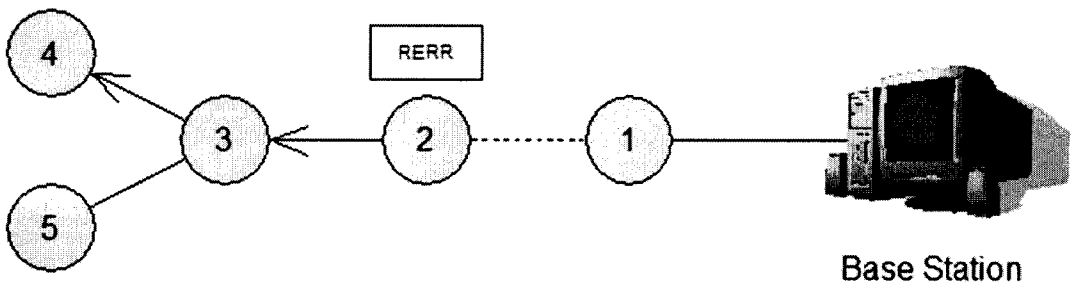


Figure 9 - Route Maintenance

An example of route maintenance can be found in Figure 9. Here, assume that Node 4 is sending data to the base station via the stored route 4-3-2-1 (found during the Route Discovery process illustrated in the previous example). Now assume that the link between Nodes 1 and 2 fails, rendering the stored route invalid (denoted by the dashed link in Figure 9). The next time Node 2 receives a message sent by Node 4, it will check the route record stored in the message to discover the next hop (Node 1). After sending the message to Node 1 and never receiving an acknowledgement, Node 2 will generate an

RERR message stating that the message could not be forwarded due to a failed link. The RERR is then unicasted back to Node 4 through Node 3.

2.3.3 Advantages and Disadvantages

DSR has many advantageous properties that make it well suited to networks with a highly dynamic topology. Perhaps the greatest benefit of DSR routing is that only the routes a node will actually use are stored. In other ad-hoc routing protocols, such as DSDV [12], each node is required to store route information for all reachable nodes in the network, resulting in potentially large routing tables. Since all routes in DSR are discovered and stored on-demand, we minimize the amount of memory required for a node to store routing information. This is a huge benefit in sensor networks where nodes have a scarce amount of memory to allocate for such functions.

Another advantage of DSR is that there exists no requirement for periodic routing update messages. Broadcasting such messages increases the energy strain on all nodes in the network, resulting in a reduced network lifetime. Since DSR is on-demand, there is no need to broadcast routing update messages unless absolutely necessary (i.e., – sending an RRQ).

A third benefit of DSR routing is that it supports unidirectional links. Such an approach is beneficial in sensor networks since not all links may be bidirectional. Support for unidirectional links is made possible because the destination of an RRQ can also initiate a separate route discovery to the source, resulting in a (possibly) different return path. As a result, the base station can potentially increase its transmission range so that messages can reach all nodes in the network in one hop; however, any data being sent back to the base station may have to be routed through many intermediate nodes.

An additional benefit of DSR relates to the fact that several paths can be stored to route data back to the base station. This allows a data generating node to periodically switch the current path being used in order to minimize variation in energy consumption. After all, if the same routing path were used continuously, the nodes on that path would consume energy more quickly than other nodes in the network. This would result in an earlier failure of nodes on the routing path due to battery depletion, thereby negatively impacting the performance of the network.

A disadvantage of DSR is that each packet sent in the network must contain the sequence of hops required to reach the destination. This significantly increases the per packet overhead.

Also, if the network topology is extremely dynamic, source routes will have to be re-discovery frequently. This means that RRQ messages will be constantly flooding throughout the network, thereby increasing the strain on the energy supply of all nodes. As well, a dynamic topology increases the delay in transmitting packets to their destination since the sending node must wait to obtain a valid source route.

2.4 Minimum Transmission Routing

Minimum Transmission Routing is the last routing protocol examined in the thesis. This protocol takes a distance vector approach to routing and selects paths based on the estimated number of transmissions necessary to reach the source (i.e., the base station) [25]. As well, Minimum Transmission routing utilizes a tree-based routing topology that is updated based on periodic routing messages. This implies that any data that needs to be routed will always be sent through a node's current parent. Neighbour discovery is

accomplished both through passive monitoring (overhearing messages sent to another node) and through beacon messages that are transmitted periodically to aid in topology formation. Minimum Transmission routing also uses a fixed-sized routing table to store neighbour routing information where entry into the table is governed by insertion, eviction, and reinforcement policies.

2.4.1 Routing Table Management

Since the Minimum Transmission routing protocol uses a fixed-size routing table, there may only be space for a subset of a node's neighbours in the table. Each table entry contains routing data (e.g., number of hops to the base station for this neighbour), link estimation data, and a frequency count (used in the reinforcement and eviction policies).

When a message is received from a neighbour whose information is not currently stored in the routing table, the insertion policy will add this neighbour's information to the table with a certain probability. In order to maintain stable routing information, the rate of insertion into the table must be lower than the rate of reinforcement [25]. This helps to ensure that neighbours whose information is already in the routing table have a chance to be reinforced before their entries are evicted to make room for new insertions. The authors of the Minimum Transmission routing protocol choose an insertion probability of

$$P = T / N$$

Where T is the number of entries in the node's routing table and N is the number of distinct neighbours (an estimated value). Therefore, when a message arrives from a neighbour whose information is not stored in the table, that neighbour will be added to

the routing table with a probability of P. If a message arrives from a neighbour that is already stored in the routing table, the protocol will reinforce this neighbour using the method described below.

The reinforcement and eviction policies use the FREQUENCY algorithm defined in [26]. This algorithm maintains a frequency count for each neighbour in the routing table. When a message is received from an existing neighbour, its entry is reinforced by incrementing the frequency count in the routing table entry by one.

If the protocol decides to add a non-existing neighbour to the table (using the probability, P, defined above), and the routing table is full, the eviction policy must be run in order to free up table space [25]. To do this, the routing table is examined to find any entries that have a frequency count of zero. If such an entry is found, it is removed from the table and a new table entry is created for the non-existing neighbour. If no entries with a frequency count of zero can be found, however, each entry in the table has its frequency count decremented by one and the non-existing neighbour will not be added to the table. This policy helps to ensure that neighbours who are frequently heard from stay in the routing table, as their frequency counts are continuously updated.

2.4.2 Link Estimation

Once neighbours are stored in the routing table, the Minimum Transmission routing protocol begins to record link reliability estimations for each neighbour. In selecting a link estimation technique, the authors of this protocol investigated several possibilities and settled on using the Window Mean With Exponentially Weighted Moving Average (WMEWMA) technique. WMEWMA estimates link reliability using the formula:

L_R = (Packets received in t) / MAX(Packets expected in t, Packets received in t)

Where t is an interval in time and L_R is the link reliability. As the formula indicates, link reliability is based on the number of received/expected packets over some interval of time. Since the Minimum Transmission routing protocol assumes that each packet sent in the sensor network contains a source ID and link sequence number, the expected number of packets can be determined by comparing the current link sequence number against the stored link sequence number. For instance, if a packet is received from a node, X, with a link sequence number of 10, and the last recorded sequence number from neighbour X is 4, there were presumably 6 transmissions from node X that were not heard. Therefore, over this time interval, the expected number of packets is 6 while the number of packets received is 0.

2.4.3 Tree-Building Algorithm

As mentioned above, the Minimum Transmission routing protocol uses a tree-based topology as the framework to route messages from a source to the sink (base station). Each node in the network periodically broadcasts a routing update message to all its neighbours that contains the node's address and its routing cost to the sink (the base station always has a routing cost of 0). When a node receives a routing update message, it extracts this information and places it in the routing table entry associated with the sending node. After the routing table is updated, the node runs a parent selection algorithm to select the best parent for this node. In essence, the node in the routing table with the lowest routing cost is selected to be the parent. Once a parent has been selected, the node will update its own routing cost using the formula:

$$C_N = C_P + C_L$$

Where C_N is the node's routing cost, C_P is the routing cost of the node's parent, and C_L is the cost of the link between the node and its parent. The link cost (C_L) is defined by the following formula [25]:

$$C_L = (1 / L_{R\,(forward)}) * (1 / L_{R\,(backward)})$$

Where $L_{R\,(forward)}$ is the link reliability of the outgoing link (i.e., toward the node's parent) and $L_{R\,(backward)}$ is reliability of the incoming link (i.e., toward the node's child). The next time the node is scheduled to send a routing update message, it includes the new routing cost in this message and broadcasts it to all neighbours.

Figure 10 shows an example of a tree-based routing topology. The numbers on the links represent the link cost (C_L) and the numbers in the circles depict the node's id.

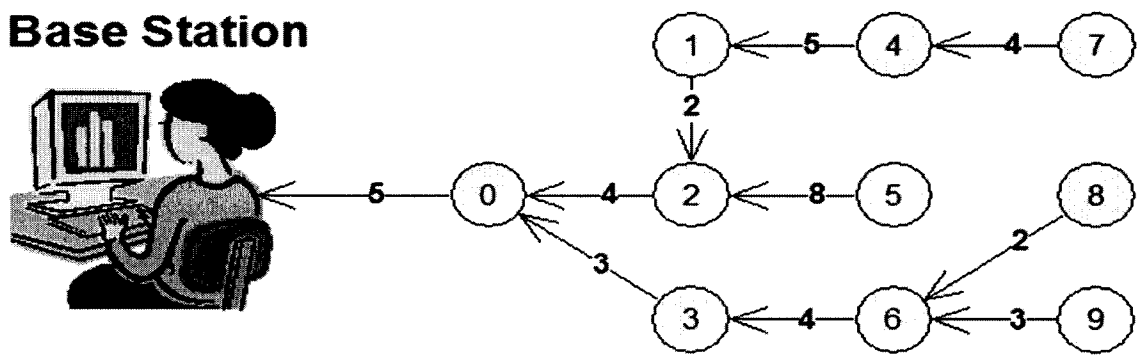


Figure 10 - Tree-based Topology

To calculate the routing cost of node 4, the formula above is used. Here, C_P is the routing cost of the parent, which is node 1. To compute node 1's routing cost, simply add the link costs from the base station down to node 1. This gives a routing cost of 11 for node 1, so C_P is 11. C_L is the cost of the link between node 1 and node 4, which is 5. Therefore:

$$C_4 = C_P + C_L = 11 + 5 = 16$$

So, in this tree topology, node 4 would have a routing cost of 16.

2.4.4 Routing Architecture

Now that the various building blocks of the Minimum Transmission routing protocol are explained, a description of the overall architecture is presented to highlight the interactions between the various components.

Shown in Figure 11 is the system architecture of this routing protocol. When a message is received by a node, it is processed by both the Table Management component (detailed in section 2.4.1) as well as the link estimator component (described in section B). A Cycle Detection component also processes the message in an effort to detect and eliminate routing cycles. If a cycle is detected, the Parent Selection component (discussed in section 2.4.2) is called to choose a new parent. The Timer component is responsible for triggering the periodic parent selection process.

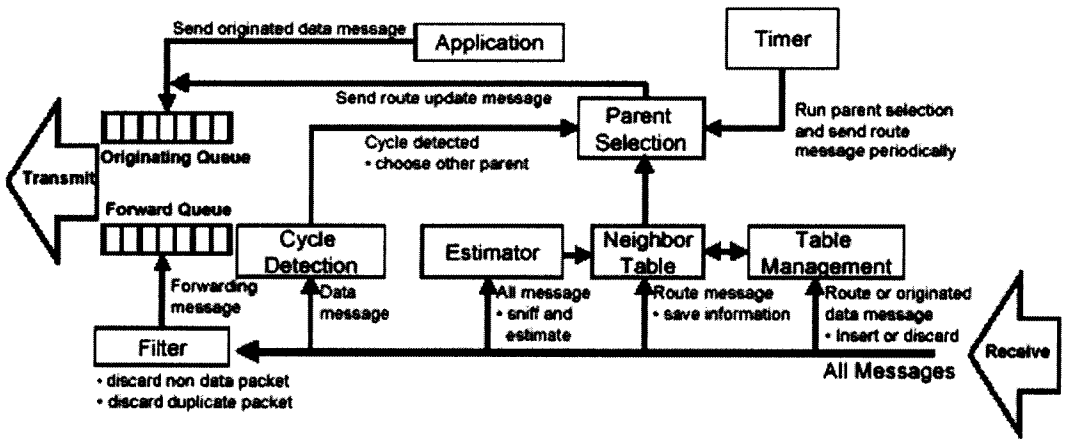


Figure 11 - Minimum Transmission Routing Protocol Architecture [25]

When an incoming data message is received, the Filter component determines whether or not this message needs to be forward towards the base station. Routing messages are not

forwarded and are simply discarded when processed. If the data message needs to be forwarded, it is placed in a Forward Queue (FIFO) to be sent. It is important to note that, since Minimum Transmission routing assumes a tree-based topology, all data messages are forwarded to the node's parent.

When the application generates data to be sent back to the base station, a similar process is followed. The message is placed in an Originating Queue (separate from the Forward Queue) to be sent.

2.4.5 Advantages and Disadvantages

One of the major advantages of Minimum Transmission routing is that it uses the minimum number of expected transmissions as a cost metric. Due to the vast energy constraints facing sensor nodes, coupled with the fact that sending messages is the major source of energy usage, this routing cost metric is ideal for sensor networks. By minimizing the number of transmissions needed to route a message to the base station, this protocol promotes energy conservation among the nodes in the network. As a result, the network lifetime is prolonged.

Another advantage of using Minimum Transmission routing is that it requires a fixed-size amount of memory to store routing information. By storing only a fixed number of neighbour entries in the routing table, the protocol maximizes the amount of memory available for use by the data collecting application.

Using a tree-based routing topology is also advantageous because it is designed for a many-to-one routing scenario. In the majority of habitat monitoring applications,

this scenario is quite relevant as all sensor nodes in the network will be transmitting data back to only one destination – the base station.

An obvious disadvantage of Minimum Transmission routing is that it requires periodic broadcasting of routing messages by all nodes. In contrast to a protocol such as DSR (which requires no periodic routing messages), broadcasting these messages puts strain on the energy supply available to nodes in the network. As well, even if a node is not actively sending data back to the base station, it must still generate, transmit, and process routing update messages.

Another disadvantage of Minimum Transmission routing is that it requires passive participation for all nodes in the network. This implies that nodes must continually monitor network traffic to snoop on packets that are not destined for them. Since a node must process each packet it hears, the node has a smaller chance of entering a sleep state to reduce battery consumption. Having all nodes in the sensor network active for the majority of the time can substantially decrease the network lifespan.

A further disadvantage of Minimum Transmission is the potential loss of data messages when a network error occurs. By design, every node in the Minimum Transmission protocol updates the routing cost for each of its neighbours at a regular interval. Thus, if a node fails, its neighbours should notice this during the next scheduled routing update and switch to a new parent accordingly. However, if the routing update interval is not sufficiently small, it is possible that data messages will be lost. For example, if the routing update interval in 10 seconds but the data generation interval is 5 seconds, 2 data messages may be lost before a new parent is selected. In some sensor network deployments (e.g., object tracking), any loss of data can have disastrous effects.

2.5 Tiny Microthreading Operating System (TinyOS)

TinyOS is a lightweight, event-based operating system for wireless sensor networks and other embedded devices [15]. It was designed for the ATMEL AVR family of microcontrollers but has since been ported to other architectures. TinyOS is completely open source, quite stable, has a large user base, and an active support community. Due to these reasons, TinyOS is the operating system chosen for use in the experimentations.

TinyOS was developed to adhere to certain principals applicable to tiny embedded devices such as sensor nodes. Perhaps the main motivation behind TinyOS is making an operating system that is both energy and memory efficient. To achieve energy-awareness, TinyOS adopts an event-driven design to maximize the amount of time that a sensor node remains in an idle, or sleep state. TinyOS was also designed to use as little memory as possible. On an ATMEL ATMEGA103L processor, a complete sensor network application occupies only 3K of instruction memory and 226 bytes for the operating system's data store in SRAM.

Another design goal of TinyOS is to support concurrency-intensive operations [15]. In sensor nodes, it is likely that operations may be taking place simultaneously. For instance, a temperature sensor may be collecting readings at the same time as the radio is receiving an incoming routing message. The event-driven nature of TinyOS allows concurrency to be achieved in a small amount of space.

TinyOS is written is a fairly new language called nesC, which has a C-like syntax but is more suited towards embedded systems [30]. In nesC, the main software concepts are interfaces and components. Interfaces dictate a set of functionality that must be provided by any software components choosing to implement this interface (much like the interface functionality in Java). A component is a basic software module that

provides and uses interfaces. For example, a sensor interface may require that components implementing the interface provide functionality for turning the sensor on and off.

Commands, events, and tasks are the building blocks from which components are constructed. Commands are non-blocking methods that can be called by the interface user and are implemented by the interface provider. Events occur in response to hardware interrupts and must be handled by the interface user. This is done by having the calling component define event handlers that are triggered in response to specific events.

Tasks are the most basic building block for creating components and are responsible for performing most of the work. Essentially, tasks are methods whose execution can be deferred. When a task is called, it is placed in an internal FIFO task queue that determines processing order. Once a task begins execution, it must run to completion before the next task can be executed. Although this means that tasks cannot be pre-empted by other tasks, they can still be pre-empted by event handlers.

Applications written for use with TinyOS are most often written in the nesC language. TinyOS applications are comprised of two main components: modules and configurations. Modules are where the application code resides, including the implementations for any interfaces used by the program. The configuration file dictates how the modules in the application are connected, or wired. It provides a matching between interfaces used by components and interfaces provided by components. This allows a user to change which implementation of an interface a component uses without

changing any application code. All that is needed is to change the association in the configuration file to point to a different implementation.

For an example of this, consider the case where a user decides they want to gather soil moisture readings as opposed to temperature readings. The line in the configuration file that wires the interface to the temperature sensor implementation may look like:

```
MyAppM.ADC -> TemperatureSensor.ADC;
```

This line maps the Analog to Digital Converter (ADC) interface used by the MyAppM component (a sample application) to the implementation provided by the TemperatureSensor component. To change the implementation to use a soil moisture sensor instead, the above line can be changed to:

```
MyAppM.ADC -> MoistureSensor.ADC;
```

As is evident, the user can simply edit the configuration file to specify which sensor implementation their module should use. This is a good example of the modular design of TinyOS. By re-wiring components in the configuration file, it is possible to dramatically alter the operation of an application without touching any application code.

2.6 TOSSIM

TOSSIM is a simulator for TinyOS-based sensor networks that can accurately and efficiently simulate networks containing thousands of nodes [16]. As such, it is a vital tool in the development of TinyOS-based sensor networks because it allows for testing and analysis of applications before deployment occurs. As well, since it is infeasible in most cases to obtain a large number of actual sensor nodes, TOSSIM provides researchers an opportunity to see how their work scales to hundreds or even thousands of

nodes without requiring the physical hardware. Due to the experimentations in this thesis requiring a fairly large number of nodes (approximately 40), TOSSIM is well suited to our needs for the above reasons. As such, TOSSIM is used as the platform on which our experiments are run.

TOSSIM was designed with four specific requirements in mind [16]. The first requirement is that the simulator must be scalable such that it can handle a network containing thousands of nodes. The second requirement is completeness, meaning that the simulator must model the actual system behavior and interactions as closely as possible so that simulation of an application is a good approximation of how those applications will behavior on actual hardware. The next design requirement is fidelity, which states that the simulator must properly encapsulate the network behavior at a very low level. Network communications are perhaps the most important aspect of sensor network applications so accurately capturing the behavior of the network is essential. The final requirement is bridging, which refers to the validation of algorithm implementations. It is crucial that developers can use the simulator to determine whether or not their implementations will run correctly on real hardware. It is also important to note that the TOSSIM researchers performed various experiments to ensure that each of the design requirements of TOSSIM was being validated. These results can also be viewed in [16].

TOSSIM executes the same code as the actual sensor hardware [16]. To accomplish this, several of the underlying TinyOS hardware abstraction components are re-written to support the simulator framework. Examples of such hardware include the Analog-to-Digital Converter (ADC) and the EEPROM. The designers of TOSSIM also

made changes to the nesC compiler included with the TinyOS distribution such that application code can be compiled directly into the TOSSIM framework. The result of the compilation is a single executable that contains both the application code as well as the TOSSIM framework code.

Additionally, TOSSIM implements an internal event queue. When one of the hardware abstraction components generates an interrupt, it is placed in the event queue. When an interrupt in the event queue is being served, the queue triggers the appropriate event-handler in the application code to handle the event.

Another important property of TOSSIM is that it simulates the TinyOS networking stack at the bit level [16]. As well, TOSSIM provides both a simple and a lossy radio model. In the simple radio model, all nodes are within communication range of each other and each transmitted bit is always received at the destination. Collisions are still possible, however, and may result in corrupted packets. The lossy model allows the user to define a directed network graph where two nodes can communicate with each other only if there is an edge between them. Also, each edge in the graph has a bit error probability attached to it that determines whether or not a bit is corrupted during transmission (bit errors are independent). These directed graphs are defined in a topology file where each line is of the format:

<node id>:<node id>:bit error rate.

For example, if node 1 can hear transmissions from node 0 with a bit error probability of 5%, this is represented by the line 0:1:0.05.

TOSSIM also provides an interface to external applications via a Transmission Control Protocol (TCP) socket so that simulations can be monitored or altered on demand

[16]. For instance, this mechanism can be used to inject packets into a running simulation (e.g., interest messages). These features are used extensively in our experimentation and will be discussed more thoroughly in Chapter 4.

2.7 PowerTOSSIM

As mentioned above, energy consumption is one of the most important aspects to consider when deploying a sensor network. As such, many of the metrics selected for the experimentation deal with the energy expended during operation of the routing protocols (detailed in Chapter 3). Since TOSSIM does not contain functionality for measuring the energy consumption of simulated applications, it is necessary to use other means to record such measurements. PowerTOSSIM is a popular TOSSIM plug-in that can accurately record energy consumption of individual components (e.g., CPU, radio, etc.) on a node-by-node basis [27].

The main functionality behind PowerTOSSIM is based on power state transition messages [27]. These messages are logged whenever there is a change in the power state of a component (e.g., ADC, radio, etc.), effectively providing the length of time that each component is active over a given simulation. To facilitate the collection of these messages, each TOSSIM hardware representation is connected to a PowerState component that is invoked whenever a power state transition occurs.

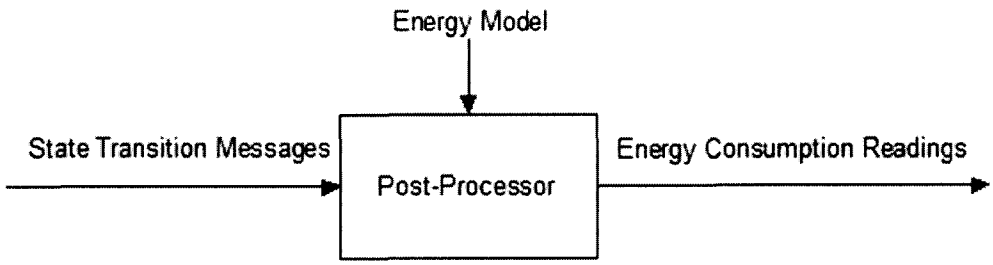


Figure 12 - Operation of PowerTOSSIM

A high-level representation of how PowerTOSSIM operates can be seen in Figure 12. The state transition messages are combined with an energy model in a post-processing module. The output of this model provides the energy consumption readings for each node participating in the simulation

The energy models calculated for use with PowerTOSSIM were obtained through extension power profiling with real sensor hardware (with the assistance of tools such as oscilloscopes). The result is an accurate simulation of power consumption that provides measurements within 0.5-13% of the readings from running the application on actual sensor hardware. Additionally, PowerTOSSIM scales well even to simulations involving upwards of 1000 nodes.

Since PowerTOSSIM does not simulate the CPU of each sensor node at the instruction level, the authors use an approximation technique to determine the power consumed by the CPU. To start, the TOSSIM binary is modified to associate a counter with every basic block of code. This counter is incremented each time the basic block is executed. A basic block is defined as a non-branching set of continuous instructions. Each basic block is then associated with its equivalent set of AVR assembly instructions. Next, the number of CPU cycles required for each basic block is computed using simple instruction analysis. Finally, the counter values for each basic block are combined with

the CPU cycle analysis to estimate the CPU cycle count for each node in the simulation. Since the above processing is fairly involved, the CPU cycle analysis takes place after the simulation has finished. Although the process is not perfect, the readings gathered are fairly accurate for most applications, falling within a 3% error margin. However, during their evaluation, the authors found that a small number of applications had a margin of error of up to 33% (e.g., the quicksort sorting algorithm). The authors attribute this discrepancy to inaccurate cycle mappings for a small number of basic blocks.

2.8 Tython

During the course of the experimentations, it is necessary to introduce specific node failures into a running TOSSIM simulation. This is done to help evaluate how well a routing protocol adapts to a network error and is explained further in Chapter 3. To interact with the running simulation, a TOSSIM extension called Tython is used. Tython is a scripting interface based on the Python language that allows users to manipulate the current state of a running simulation [28]. Using Tython, a user can employ simple scripting commands to perform such actions as stopping, starting, or pausing the simulation, dynamically changing the physical location of nodes in the network, or turning nodes on or off at any point during the simulation. Tython is able to communicate with TOSSIM by use of the TCP socket mentioned in section 2.6.

In the experimentations, Tython serves only a minor purpose. At some point during an error simulation, when it is decided that a particular node should fail, Tython is used to pause the running simulation. Another Tython scripting command is then used to turn off the desired node. Finally, a Tython command is invoked to resume the

simulation after the node has been turned off. The effects of introducing the network failure are then measured based on the metrics defined in Chapter 3.

Chapter 3 – Experimentation Plan

3.1 *Motivation*

In order to effectively compare the selected routing protocols, it is necessary to test them under a common set of conditions. By measuring the performance of each protocol in relation to well-defined metrics, a proper evaluation can be achieved. Once the experimentation plan has been carried out, the results can be analyzed to decide which routing protocol performs most desirably for each metric and to determine in which circumstances the protocol performs the best.

3.2 *Metrics*

To facilitate the experimentations, a set of metrics has been developed against which the routing protocols will be compared. These metrics are chosen due to their relevance in habitat monitoring applications. A summary of the chosen metrics can be seen in Table 3 including which scenarios they are relevant to. The Perfect scenarios entail that no errors occur in the network during the experiment. The Error scenarios introduce specific errors into the network to determine how well each routing protocol can adapt. These scenarios will be discussed in more detail later in the chapter.

Metric	Scenarios	Summary
Average Dissipated Energy	Perfect	Ratio of energy consumed by all nodes to the number of unique data messages received by the base station.
Average Delay	Perfect	Time required for a data message to travel from a source node to the base station.
Energy Consumption Variation	Perfect	Variation of energy consumption among all nodes in the network after a specified period of time.
Code and Data Size	Perfect, Error	Amount of instruction and data memory required by the routing protocol.
Total Number of Messages Transmitted	Perfect, Error	Number of messages sent during the simulation.
Recovery Total Energy Consumption	Error	Total amount of energy consumed by all nodes during the time required for the network to stabilize after an error.
Number of Data Messages Lost	Error	The number of data messages lost during a network error.
Stabilization Time	Error	Time required for the routing protocol to stabilize after an error has occurred.

Table 3 - Summary of Chosen Metrics

The first metric devised for use in the experiments is Average Dissipated Energy. This metric is defined as:

$$\text{Average Dissipated Energy} = E / N$$

Where E is the sum of the energy consumed by all nodes and N is the number of unique data events collected by the base station. In essence, this metric allows us to determine roughly how much total energy is required for one data event to be generated and received by the base station. Obviously, lower values are preferred for Average Dissipated Energy. A low value means that the base station will receive a higher number of data events before network failure occurs (due to node battery exhaustion).

Another metric used in the experimentation plan is Average Delay, or the average time it takes for a recently generated data event to reach the base station. Delay is defined as:

$$\text{Delay} = T_1 - T_0$$

Where T_1 is the time at which the data event reaches the base station and T_0 is the time at which the data event is sent by the source node. The Average Delay is then calculated by taking the average of the delays associated with all data events. In some habitat monitoring applications (e.g., object tracking), it is important to receive data events as quickly as possible so that proper actions can be taken. By achieving a low Average Delay, a sensor network routing protocol is well tailored to this type of usage.

Additionally, we consider variation in energy consumption to be an important metric. This metric measures the distribution of energy consumption amongst all the nodes in the network after a specified period of time. This is an important metric because an uneven distribution will cause some nodes to fail more quickly than others. If a small subset of nodes fails early in the lifecycle of the network, it could have disastrous consequences. Thus, variation in energy consumption can be linked directly to sensor network lifetime. A smaller value for this metric is preferred because it implies that the energy consumption is more evenly distributed amongst all nodes in the network, resulting in a potentially higher network lifespan.

Another pertinent metric is the total number of messages required to be transmitted for correct operation of the routing protocols (over a given period of time). As mentioned above, radio activity is the major source of energy depletion in the sensor nodes. Therefore, routing protocols that have a low message transmission overhead

provide an obvious benefit in terms of energy consumption. Although this metric is similar to Average Dissipated Energy, it focuses solely on energy consumed through radio activity.

Recovery Total Energy Consumption is the first metric that is relevant only in the error scenarios. This metric measures the total energy required among all nodes to recover from a network error. Obviously, a lower value for this metric is preferable, especially if errors occur very frequently in the network.

An additional metric that is unique to the error scenarios is Stabilization Time, or the time taken for the routing protocol to stabilize after a network error has occurred. Routing protocols that have a higher Stabilization Time run the risk of losing data messages sent in the network. Therefore, a lower Stabilization Time is obviously a preferred characteristic.

Another metric that is relevant only in the error scenarios is the Number of Data Messages Lost when an error occurs in the network. This metric is closely related to the stabilization time and is equally important. A routing protocol should be able to stabilize while minimizing the amount of data messages lost in the process. Data messages can be lost when there is no explicit end-to-end reliability protocol in place (e.g., the Transmission Control Protocol used on the internet). This is the case in sensor network routing protocols such as Directed Diffusion. If a message is lost en route to the base station, there is no mechanism in the protocol to detect the error and retransmit the message. Also, even if a protocol does have end-to-end reliability, it is still possible to lose data messages. In Dynamic Source Routing for example, outgoing messages can be buffered while waiting for an RRP to arrive with a new path to the base station. If data is

being generated too quickly, however, this buffer may overflow before the RRP is received, resulting in lost data messages. Buffer sizes also have to be kept modest due to the memory constraints in the sensor nodes.

The last metric considered in our experimentations is Code Size. This metric is divided into two components and involves the amount of memory required in the sensor nodes to facilitate the routing protocols.

The first component of this metric is Program Code Size. Program Code Size is the amount of memory (in KB) occupied by the program instructions. Since sensor nodes typically have only a small amount of memory available to applications, keeping the size of the compiled routing protocol to a minimum is essential.

The second component of the Code Size metric is Data Size. Aside from the memory required to store the compiled code, it is also important to determine an upper bound on how much memory the routing protocols require during runtime (e.g., memory for routing tables). Only by investigating both aspects of Code Size can a conclusion be reached on the memory-efficiency of each protocol.

3.3 *Measuring Metrics*

Now that the metrics used to evaluate the performance of each routing protocol are specified, it is important to discuss how each of these metrics is measured. To measure the first metric, Average Dissipated Energy, a simulation is run for a specified period of time using the PowerTOSSIM plug-in mentioned in Chapter 2. As well, each time a data message is received by the base station, a log message is generated to indicate that fact. At the end of the simulation, the output of the PowerTOSSIM plug-in is analyzed to obtain the total amount of energy expended by all nodes. This value is divided by the

total number of unique data messages received by the base station during the simulation (also determined by examining the application log file). The Average Dissipated Energy is the resultant value of the above operation.

A custom-developed Java program, PowerLogTool, is used to examine the output of the PowerTOSSIM plug-in and compute the Average Dissipated Energy. An example of the output from the PowerTOSSIM plug-in can be seen in Figure 13. For each sensor mote (i.e., node), the total power (in mJ) used by both the radio and CPU is listed. The PowerLogTool program iterates over the output, adding up the power consumption totals for all nodes. This value is then divided by the number of data messages received to produce the Average Dissipated Energy.

```
Mote 0, radio total: 2.480016
Mote 0, cpu_cycle total: 1.159321

Mote 1, radio total: 1.666896
Mote 1, cpu_cycle total: 1.096799

.....

Mote 35, radio total: 19.438650
Mote 35, cpu_cycle total: 1.721442
```

Figure 13 - Sample PowerTOSSIM Output

When using the PowerTOSSIM plug-in to compute energy consumed by the radio, only the energy required to send messages is considered; the energy required to receive messages is disregarded by setting a parameter in the PowerTOSSIM configuration file. This provision exists because the radio implementation used by TOSSIM assumes that the radio is always in receive mode, sampling the radio channel for incoming messages. Since, in this model, sending and receiving messages consumes

roughly the same amount of energy, the energy required to send messages is swamped by the energy used in receiving them. For example, in a 3 minute simulation, the radio expends approximately 3000 mJ receiving messages while sending messages requires less than 500 mJ (depending on the routing protocol and scenario being used). This is explained by the fact that the radio is only be in the sending mode for a few seconds but is in the receiving mode for the entire simulation. Since the energy used for receiving messages is a constant among all protocols, it can safely be ignored. By focusing only on the energy required to send messages, a better understanding can be gained as to which routing protocols are more energy efficient.

To obtain measurements for the Average Delay metric, each routing protocol implementation is instrumented to generate a time-stamped log entry whenever a data message is sent by a data-generating node. Similarly, each data message received by the base station is associated with a timestamp value to indicate arrival time. The delay for a single message is determined by computing the difference between these two timestamps. This computation is done for each data message transmitted throughout the simulation lifetime and the Average Delay is the resulting average value among all these computations.

Similarly to the Average Dissipated Energy, the Average Delay is calculated using a custom Java program called DelayLogTool. DelayLogTool scans through the simulation log file to find log statements indicating that the data generator has sent a data message. This entry is then matched to another log entry indicating that the base station has received the transmitted data. Since timestamps are associated with both log entries, a delay is calculated using a simple subtraction. The Average Delay is determined by

taking the average of all calculated delays. An example of the log statements used in this analysis can be seen in Figure 14. For example, the first statement in this log snippet states that node 35 is sending data with a unique id of 665 at a time of 36.08 s. The second statement indicates that node 0 (i.e., the base station) receives this data message at a time of 36.35 s.

```
35: APP: Data ready to be sent: (665) (0:0:36.07885275)
0: Base Station received new data (665) from node 35 at(0:0:36.34833725)
35: APP: Data ready to be sent: (491) (0:0:40.96135275)
0: Base Station received new data (491) from node 35 at(0:0:41.23074675)
.....
```

Figure 14 - Sample delay log statements

In Chapter 4, as part of the proposed software architecture, it is revealed that a media access control (MAC) layer is used to prevent collisions amongst broadcast messages. The solution entails adding a short, random delay before broadcast messages are sent. This obviously influences the Average Delay value, so to counteract the problem, the MAC waiting times are *not* included in the Average Delay metric. This is a reasonable assumption because media access control is not meant to be an important component of our experimentation analysis – the MAC layer is implemented only to solve the collisions problem.

The PowerTOSSIM plug-in is also used to measure the variation in energy consumption. At the end of the simulation, the PowerTOSSIM output is analyzed to determine the total energy consumed by each node in the network. These values are used to compute the standard deviation of energy consumption amongst all nodes in the network using the formula:

$$\text{Standard Deviation} = \text{sqrt}(\sum (X - u)^2 / N)$$

Where u is the mean of the energy consumption values and N and is number of nodes in the network.

The total number of messages required for correct operation of the routing protocol is simply calculated by counting the total number of messages sent by all nodes in the network over the course of the simulation. The application log file is again used to obtain how many messages are actually sent.

In the error scenarios, the Recovery Total Energy Consumption is measured in much the same way that Average Dissipated Energy is in the perfect scenarios. The PowerTOSSIM plug-in is used to analyze the error log file and generate power consumption statistics for each node. The power consumed by each node is then combined to create the value for the Recovery Total Energy Consumption metric. As noted above, the energy required by the radio to receive messages is not included in these calculations.

Another error scenario-specific metric, Stabilization Time, is measured by analyzing the log file and determining the times at which the network error was detected and when the first post-failure data message is received by the base station. The stabilization time is determined by subtracting these two values. It is important to note that the point at which the network error is detected varies depending on the routing protocol being used. This fact is explored in more detail in the proceeding section.

Similarly, in the error scenarios, the number of data messages lost during a network error is determined by inspecting the log files and looking for data messages sent by the data source that were not received by the base station.

To measure Program Code Size, the compiled executable for each routing protocol is examined to determine how much memory it occupies. This is done using the ncc compiler included with the standard TinyOS distribution. The command used to compile each of the implemented routing protocols is:

```
ncc -o main.exe -target=mica2 SimpleSense.nc
```

Where mica2 is the target hardware platform and SimpleSense.nc is the name of the application configuration file. Since only a single routing protocol can be specified in the configuration file at a time, this command must be executed three times in order to measure the Program Code Size for each routing protocol.

To measure Data Code Size, each routing protocol implementation is manually analyzed to determine the largest amount of stack space and global data possibly required at any point in time. This is done by evaluating the stack space necessary for each event that can occur during program execution (e.g., receiving a message, sending data, or performing routing updates) and choosing the largest value. Although this value does not include the amount of space required for saved program counters or spilled registers, these items are a constant overhead and can be safely ignored.

For a simple example of the Data Code Size evaluation, refer to Figure 15. Here, assume that receipt of an incoming message triggers the *messageReceived* method. Notice that this method declares only a single 8-bit integer meaning that it requires 1 byte of data memory. Later in the execution of this method, a call to *func2* is made, requiring the variables declared in *messageReceived* to be pushed onto the stack (1 byte). Notice that *func2* also declares an 8-bit integer, thereby requiring an additional 1 byte of data

memory. Also notice that *func2* eventually calls method *func3*, forcing the variables declared in *func2* to be pushed onto the stack as well (2 bytes are now on the stack).

```
void messageReceived(){
        uint8_t var1;
        .....
        func2();
        .....
        func3();

        return;
}

void func2(){
        uint8_t var2;
        .....
        func3();

        return;
}

void func3(){
        uint16_t var3 ;
        ......

        return;
}
```

Figure 15 - Code Size Analysis

The final method in the calling chain, *func3,* declares a single 16-bit integer (2 bytes). Combining these 2 bytes with the amount of stack space used adds up to a total of 4 bytes. Thus, the maximum amount of data memory required when the *messageReceived* method is invoked is 4 bytes. By performing this manual analysis for every event that can be invoked in each routing protocol implementation (e.g., message received, message being sent, etc.), the maximum required Data Code Size is determined.

3.4 *Scenarios*

In order to properly simulate and measure the performance of each routing protocol, it is necessary to construct scenarios that accurately represent real-world usage. As such, the experiments are carried out against a hierarchy of scenarios that include different network

topologies and error conditions. The experiment scenario hierarchy can be seen in Figure 16.

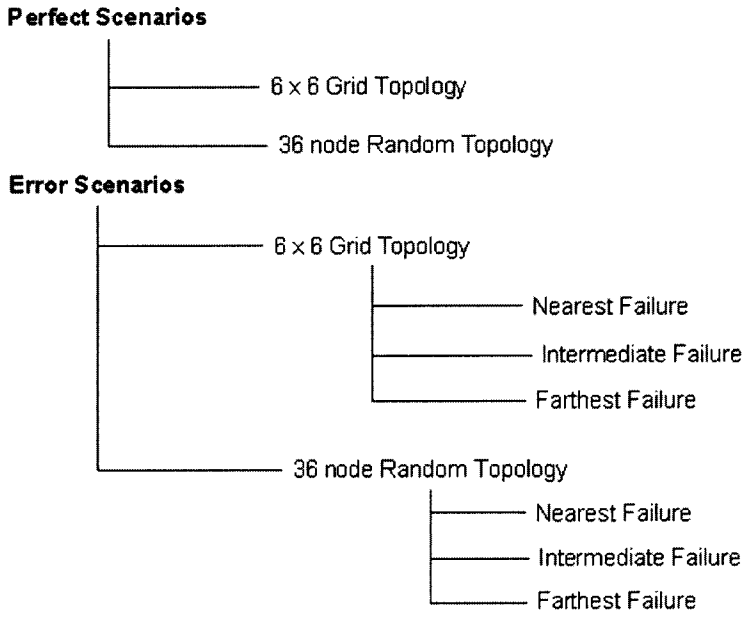


Figure 16 - Scenario Hierarchy

Note that all experiments are included under one of two possible simulation scenarios: the perfect scenarios or the error scenarios.

3.4.1 Perfect Scenarios

In the perfect scenarios, the network is impervious to any node failures. This implies that the topology of the network remains static for the duration of the simulation. Message collisions can still occur in these scenarios; however, no prolonged link failures occur. This is a reasonable assumption because, in a typical habitat monitoring deployment, the network will be placed in a location far away from common sources of radio interference such as other electronic components and metallic structures (e.g., a network located on a

forest floor or a remote island). Additionally, it allows the best possible case to be determined for each routing protocol.

In choosing the structure of the networks used in the perfect scenarios, there are two important factors to consider: the physical network topology and the number of nodes contained in the network. In choosing physical network topologies, it is important to consider the most likely sensor node positioning in habitat monitoring applications. For this reason, there are two primary topologies used in our experiments: grid topologies and random topologies. As well, each topology used in the simulations is comprised of 36 nodes. This number is chosen to represent a typical habitat monitoring network deployment such as the one described in [3]. This number is also conducive to constructing a grid.

In a grid topology, all nodes are spaced evenly to create an N x N grid. This type of network is useful in situations where the phenomena we want to measure is spread out over a uniform area (e.g., Measuring soil temperature over a particular area of interest). Thus, many habitat monitoring applications utilize a grid topology for data collection. An illustration of the 6 x 6 grid topology used in the perfect scenarios can be seen in Figure 17.

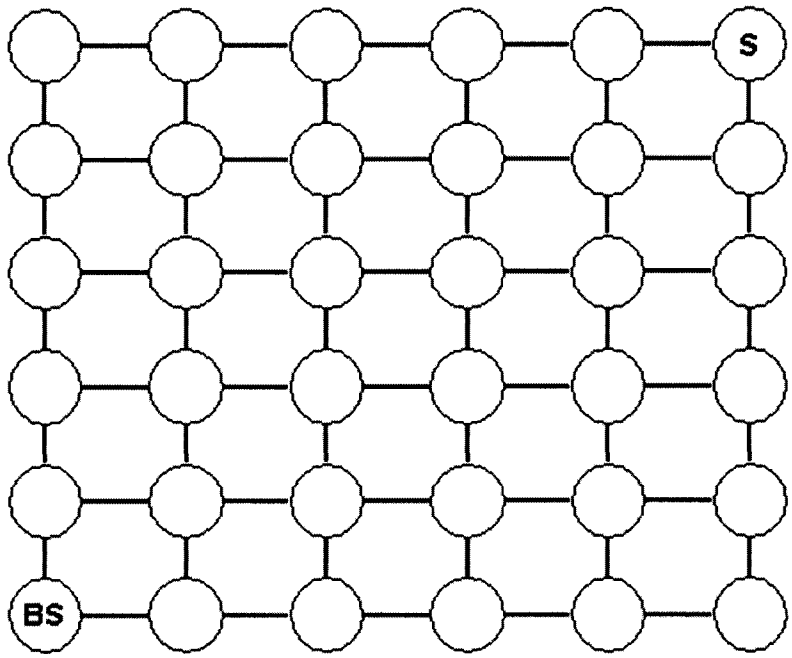


Figure 17 - 6 x 6 Grid Topology

In this perfect scenario, a single data source, node S, is responsible for generating data events and sending them back to node BS (the base station).

As the name implies, the random topology used in the perfect scenarios is a topology that is randomly generated. The motivation for using a random topology is to account for habitat monitoring deployments where the placement of the sensor nodes is dependant solely on the phenomena being studied. For example, if a researcher is interested in studying the nesting habits of birds [3], the sensor nodes must be placed wherever a burrow is found. The resulting topology obviously may not be uniform (e.g., not a grid), justifying the decision to include randomly generated topologies in the experimentations.

To facilitate the construction of random topologies, a custom Java program named TopologyGenerator is used. TopologyGenerator accepts parameters to define the number of nodes in the network, the communications radius of each node in the network (in

meters), and the physical size of the area containing the network (width and height in meters). The output of the program is a connected graph that contains the specified number of nodes arranged in a random topology. The node that is the farthest distance from the base station is considered the data-generating node. For the random topologies, a value of 36 is used for the number of nodes to stay consistent with the numbers contained in the grid topology. As well, the node's communication radius is set to 2m, which is representative of real-world values when a node is placed at ground level. Stemming from this, a network area of 12m x 12 m is used to accommodate the relatively short transmission radius.

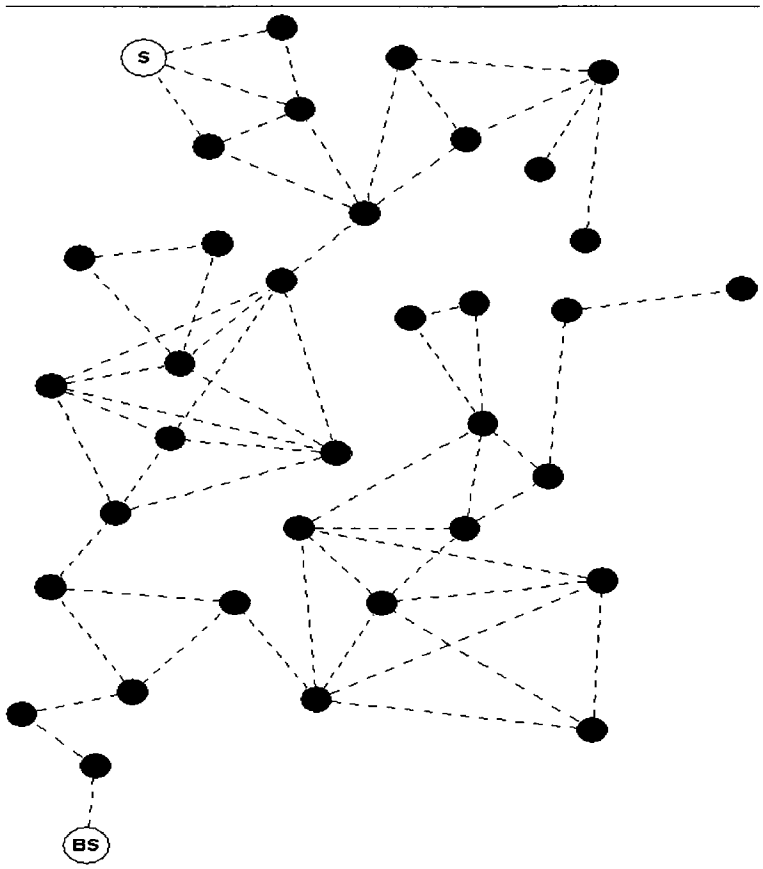


Figure 18 - Perfect Scenario Random Topology

The random network topology generated for use in the perfect scenario experiments is shown in Figure 18. As with the grid topology, node S is the data generator and node BS represents the base station.

Each of the routing protocols are simulated under both the grid and random topologies, resulting in 6 experimentation scenarios. Each scenario begins with the base station sending an interest message requesting data from the data generating node (S). This interest message requests that data be sent at an interval of every 5 seconds for a duration of 180 seconds (3 minutes). Messages then begin to flow through the network and, when the interest duration has passed, the simulation ends. After the simulation, the resulting logs are analyzed to evaluate the metrics as described above.

When running the experiments for the perfect scenarios, there are a couple of special cases that need to be handled to ensure that all three protocols have an equal chance to perform well in the metric evaluations. In the Directed Diffusion protocol, the initial interest sent by the base station is an exploratory interest that results in the base station receiving duplicate data from multiple neighbours. To ensure fairness, the initial interest sent in the Directed Diffusion simulations has duration of only 30 seconds. Otherwise, multiple copies of each data message would be sent by the data generator and propagate throughout the network for the entire simulation, resulting in unneeded energy consumption. Before the initial interest expires, the base station performs interest reinforcement (as described in Chapter 2), resulting in the base station receiving data events from only a single neighbour for the remainder of the simulation.

Another special case that is considered relates to the Minimum Transmission routing protocol. By design, this protocol requires an initial stabilization period before

messages can be properly routed in the network. If an attempt were made to send data messages immediately, they would be lost until the stabilization period ended, resulting in an unfair situation for the Minimum Transmission protocol. To accommodate this, the evaluation of the Minimum Transmission routing protocol is divided into two parts: the Stabilization Phase and the Interest Phase. The Stabilization Phase encompasses the time from which the simulation starts to the point at which the routing topology has stabilized and is able to properly route messages. The Interest Phase occurs immediately following the Stabilization Phase and begins with the base station sending the interest message as described above.

For the analysis of the Stabilization Phase, only two of the chosen Perfect Scenario metrics are considered: Energy Consumption Variation and Total Number of Messages Transmitted. An additional metric, Total Energy Consumption, is used in the analysis of the Stabilization Phase to determine the total amount of energy required amongst all nodes during stabilization. Both Average Dissipated Energy and Average Delay cannot be used in the Stabilization Phase analysis because no data messages are actually sent during this period.

After the simulation has ended, the results from both the Stabilization and Interest phases are analyzed separately to determine how well the Minimum Transmission protocol performs with respect to the given metrics.

3.4.2 Error Scenarios

The second set of experiments fall under the error scenarios category. Although the perfect scenarios provide a good idea of how each routing protocol performs under

normal operation, it is important to understand how a protocol copes with errors that occur in the network. One of the most serious and commonly occurring errors in a sensor network is the complete failure of a sensor node. If a node's battery becomes depleted or the node otherwise malfunctions, it can have a detrimental effect on the routing topology. As such, the error scenarios are devised to gauge how efficiently a protocol is able to handle these types of errors. In other words, it is important to measure the cost of a network failure in terms of each of the routing protocols.

The network topologies used in the error scenarios are chosen similarly to the perfect scenarios. Therefore, both a grid topology and a random topology are used to evaluate how well each routing protocol deals with network errors. The grid topology is a 6 x 6 grid and is identical to the one described above. The random topology, however, is re-generated to accommodate the error scenarios.

When creating a random topology for use in the error experimentations, it is important that there be at least two node-disjoint paths between the base station and the data generating node. That is, two paths must exist that do not contain any of the same sensor nodes. This is important because, if there is only one path from source to destination, and a node fails on that path, there is no way to route data to the destination. As well, even if there are two paths from source to destination, but the failing node is contained in both paths, data still cannot be routed properly. By requiring at least two node-disjoint paths in our random topology, it ensures that no matter which intermediate node in the network fails, it is still possible to route messages from a data generating node to the base station. To accommodate this requirement, the TopologyGenerator Java

is augmented to generate an appropriate network topology. The topology generated by this program for use in the error scenarios can be seen in Figure 19.

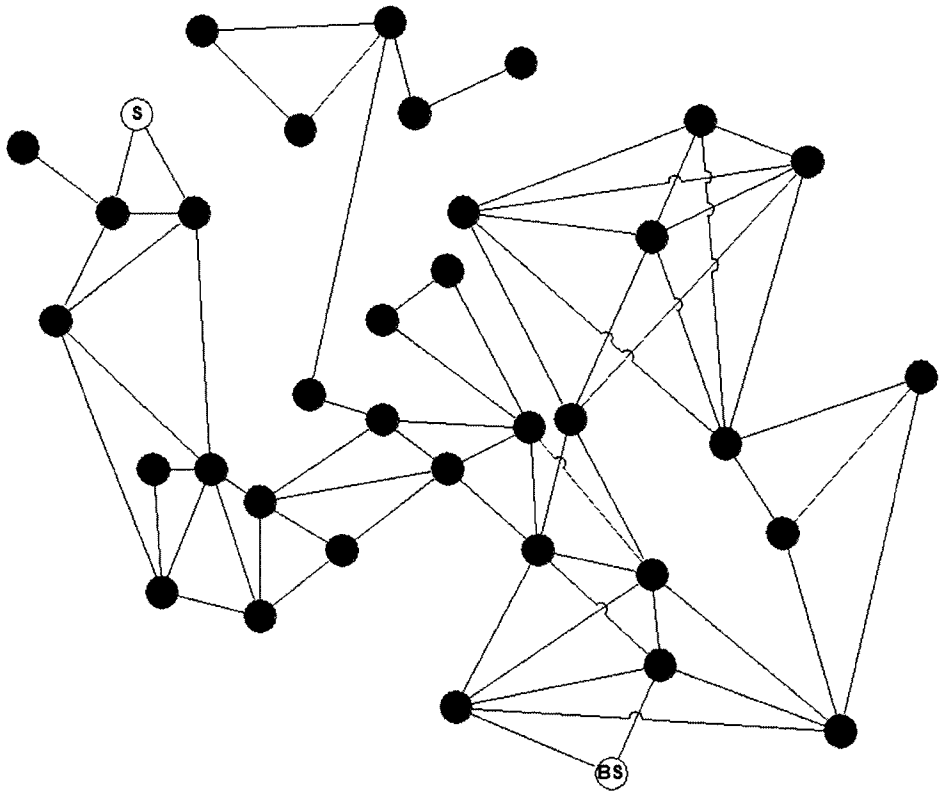


Figure 19 - Error Scenario Random Topology

Again, node S is the data generator and node BS represents the base station.

As mentioned above, the primary purpose of the error scenarios is to determine how well a routing protocol adapts to node failures in the network. When determining how to introduce node failures, it is important to remember that the position of the failing node can determine how well a routing protocol adapts to the failure. In DSR, for example, if the node failure is close to the data generator, the RERR message will arrive more quickly than if the failure was farther away. In Directed Diffusion, however, the placement of the node failure has no real impact on performance since the protocol has no implicit error detection.

To accommodate this fact, each protocol is exposed to three types of node failures during the error experimentations: closest, intermediate, and farthest failures. The closest failure is one that occurs two hops from the data generating node on the data delivery path, or the path being used to deliver data to the base station. The intermediate failure occurs in the middle of the data delivery path is calculated by using the formula:

$$X = \text{ceiling}(N / 2)$$

where N is the number of hops in the data delivery path. The resultant value, X denotes the failure node as the X^{th} node on the data delivery path (from data generator to base station). The farthest failure is one that occurs in the node adjacent to the base station in the data delivery path.

Since the data delivery path can be different depending on the topology and routing protocol used, it is necessary to have a way to determine this path dynamically. This is accomplished by examining the log file of a running simulation and manually determining which path is being used to route data back to the base station. Once the failure node has been selected, Tython is used to connect to the running TOSSIM simulation and turn off the selected failure node (a more detailed explanation of Tython can be found in Chapter 2). The following simple Tython script is used for this purpose:

```
sim.pause();
motes[x].turnOff();
sim.resume();
```

In this script, x refers to the ID of the selected failure node.

To run the error scenarios, each routing protocol is simulated under both the grid and random topologies. In addition, for each topology, there are 3 separate experiments

to capture the different types of node failures mentioned in the above paragraph. This results in 6 error experiments for each routing protocol.

In each error experiment, the routing protocol is allowed to stabilize without any network failures occurring. Once the protocol has stabilized (i.e., a single route is being used to route data back to the base station), a node failure is introduced into the network using Tython as described in the above paragraph (closest, intermediate or farthest, depending on the simulation). The simulation then continues to run until network stabilization occurs. Network stabilization is defined as the point in time when the first post-error data message arrives at the base station. The simulation logs are then analyzed between the error detection point and the network stabilization point to compute values for the error metrics.

The error detection point, or the point at which the network error is detected, differs depending on the routing protocol being used. In Directed Diffusion, there is no inherent error detection present for nodes in the network. If a node in the data delivery path fails, the only entity capable of detecting the error is the base station. If the base station does not receive the next data message as expected, it will re-broadcast another interest message to discover a new data delivery path. For example, if data is arriving at an interval of 5 seconds but the base station has not received a new message in 6 seconds, it can deduce that there is a network error. Therefore, in Directed Diffusion, the error detection point is defined to be the point in time at which the base station detects that a network error has occurred.

In contrast, Dynamic Source Routing supports error detection at the node level. This is achieved through use of the Route Error messages (RERR) that are used to notify

the data generator that a problem has occurred. Any node in the network is capable of detecting a malfunction, making error detection almost instantaneous. Based on this fact, in Dynamic Source Routing, the error detection point is defined to be the time at which a node on the data delivery path determines that a node failure has occurred.

In Minimum Transmission routing, there is no error recovery mechanism; error detection is implicit to the protocol itself. This functionality is provided by the periodic routing update messages broadcasted by each node in the network. If a node were to fail, its routing cost in relation to its neighbours would increase. This forces each neighbouring node to minimize routing costs by selecting another parent node to route data through. As such, there is no actual cost associated with a network error in the Minimum Transmission routing protocol – the routing update messages are sent regardless of whether or not errors exist. In order words, the cost of network errors is built-in to the normal operation of the routing protocol. For this reason, Minimum Transmission routing is not considered in the error scenarios but will be considered in the analysis of the perfect scenario results.

Chapter 4 – Testing Framework Software Architecture

4.1 Overview

Testing of the different routing protocols is aided by a common framework. This simplifies the implementation for each routing protocol and ensures consistent treatment during the experimentations. In this chapter, the software architecture is presented that provides a common framework for interest-based routing. As well, the importance of media access control is discussed. Finally, the experimentation setup is discussed to explain how data is collected in relation to each of the chosen metrics presented in Chapter 3.

4.2 Architecture Description

In developing a software architecture for interest-based routing, one of the most important considerations is ensuring that any protocols implemented in this architecture are sufficiently abstract such that they can be easily integrated into any target application. It should not be necessary for the application to have any knowledge of the inner workings of the routing protocols in order to use them.

Another major consideration when developing a software architecture for interest-based routing is how to actually handle interest propagation. Since an interest-based sensor network is one of the key assumptions in the experimentations, it is important that the routing component take care of managing interests. This includes broadcasting/re-broadcasting interest/reinforcement messages, maintaining an interest cache, and notifying the application when a new interest is received. The interest handling should also be abstract to the core routing algorithm implementation; however, since some

routing protocols require access to the interest cache to make routing decisions (e.g., Directed Diffusion), an interface to the interest-handling component is provided to allow access to such information.

Given the above considerations, a high-level representation of the proposed software architecture is pictured in Figure 20. The Application, DataRouter, InterestManager, and GlobalMAC entities present in Figure 20 are components. On the other hand, ReceiveInterest, InterestManager, Send, SendMsg, and ReceiveMsg are interfaces. Components contain the application logic (i.e., the code) while interfaces define how the different components are able to interact with each other. Each component and interface in the architecture is now discussed in detail.

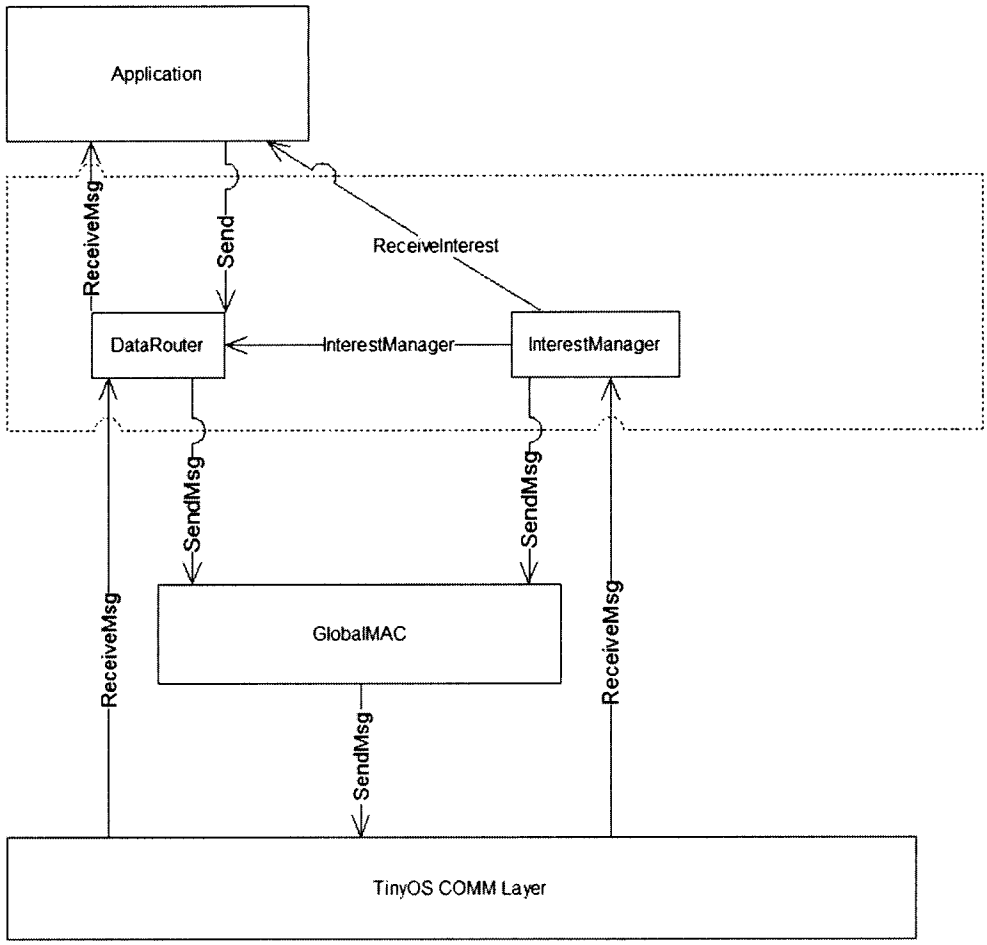


Figure 20 - Proposed Software Architecture for Interest-Based Routing

4.2.1 Application Component

The Application component represents the target application wishing to utilize any of the routing protocols. Although not directly part of the routing component architecture, there are several assumptions on the operation of the application. It is assumed that the application is interest-based. This is, interests received from the base station govern data collection by the application. As such, the necessary logic to determine whether or not an interest can be fulfilled is determined by the application and not the routing protocol. Any new interests received by the routing protocol are passed up to the application for

processing. The application will look at the criteria specified in the interest message to determine whether or not it can service it.

To utilize the software architecture, the application must implement two interfaces. The first interface, ReceiveMsg, is used by the routing protocol to pass application-specific messages up to the target application (e.g., Data messages). The second interface, ReceiveInterest, is required by the routing protocol to forward interest messages on to the application layer. When the application wishes to send a message using the desired routing protocol, it simply uses the Send interface, which comes standard with TinyOS. By utilizing this approach, the application can send messages of any type using the implemented routing protocol.

4.2.2 Interest Manager Component

The InterestManager component is responsible for the interest maintenance discussed in the above considerations. It uses the generic TinyOS RecieveMsg interface to receive both interest messages and interest reinforcement messages. If relevant, these messages are also passed to the application via the ReceiveInterest interface. The interest cache is also housed within this component as well as the logic to determine whether or not an interest message needs to be re-broadcasted. Also, when an interest reinforcement message is received, the InterestManager component is responsible for determining which neighbour should be reinforced. When an interest or reinforcement message needs to be sent, the InterestManager component uses the TinyOS SendMsg to send the message to a media access control component (GlobalMAC), which is discussed later in this chapter.

4.2.3 DataRouter Component

The actual routing protocol implementation is encapsulated within the DataRouter component. When the application has a message to send, the DataRouter component determines the next required hop required to reach the destination. The TinyOS SendMsg interface is then invoked in order to send the message over the radio. Similarly, the DataRouter component receives application messages via the TinyOS ReceiveMsg interface and can then pass the message up to the application layer or forward the message onto its next hop. Additionally, the DataRouter component is also responsible for sending and receiving protocol-specific routing messages (e.g., Route Request messages in the DSR protocol) via additional instances of the TinyOS SendMsg and ReceiveMsg interfaces.

4.2.4 InterestManager Interface

The InterestManager interface is a custom interface that is necessary to facilitate interest-based routing. As such, this interface provides the routing protocol implementation with utility methods related to the interest cache. This is useful when a particular routing protocol requires interest-related information to make routing decisions. The content of the InterestManager interface is as follows:

```
interface InterestManager {
        command uint8_t matchExists(SensorDataMsg *m) ;
        command uint8_t getNeighboursRequestingData(SensorDataMsg *m, uint16_t arr[]) ;
        command uint16_t getHighestDataInterval(SensorDataMsg *m) ;
}
```

Before describing the methods contained in the interface, it is useful to explain some concepts presented in this example. First of all, an interface defines how a particular

component can be accessed [15]. A command is simply a non-blocking method that the interface provider must implement. Finally, SensorDataMsg is the name for the custom data structure representing data to be sent to the base station.

The first method in the interface, *matchExist*, takes a data message as an argument and returns 1 (true) or 0 (false) depending on whether or not an interest exists in the cache that matches the given data.

The second method, *getNeighboursRequestingData*, is used to retrieve a list of all neighbours who have an interest matching the given data (specified in the first method parameter). The second input parameter to this method is an integer array that is used to store the addresses of each neighbour requesting the data. Finally, the return value for this method is an integer that specifies the number of neighbours who matched the given data.

The final method in the InterestManager interface is *getHighestDataInterval*. Given a data message as a parameter, this method returns the highest rate at which a neighbour is requesting data of the specified type. For example, assume neighbours A and B have an interest in the cache that matches the specified data. Now assume that neighbour A is requesting this data at an interval of every 5 seconds and neighbour B wants to receive data every 10 seconds. In this case, the method would return a value of 5. This method is useful if a routing protocol needs to determine if it should downgrade the rate at which data is being received. For instance, if a node is receiving data every 5 seconds, but the result of *getHighestDataInterval* is 10 seconds, it may want to discard every second data message received to match the requested data rate.

Not all implemented routing protocols need to use the InterestManager interface. Both Dynamic Source Routing and Minimum Transmission Routing, for example, do not use interest information when making routing decisions. Therefore, interest management is actually invisible to these protocols as received interests are sent from the InterestManager component directly to the application. Directed Diffusion, however, depends on this interface to help make routing decisions. For example, in Directed Diffusion, destinations for a message are discovered by using the *getNeighboursRequestingData* method. Also, to determine the rate at which data should be sent, Directed Diffusion uses the *getHighestDataInterval* method. Finally, to determine whether or not a data matches any cached interests, Directed Diffusion utilizes the *matchExists* method.

4.2.5 ReceiveInterest Interface

The second custom interface used to enable interest-based routing is the ReceiveInterest interface. This interface must be implemented by the Application component and its structure is as follows:

```
interface ReceiveInterest {
        event void receiveInterest(Interest interest);
}
```

The ReceiveInterest interface has only one method, *ReceiveInterest,* which contains the application logic necessary to deal with an incoming interest. This method is actually an event handler (denoted by the event keyword) that is invoked when a particular event occurs (i.e., an interest message is received) [15]. As such, event handlers must be implemented by the interface user. The only input parameter to the *receiveInterest*

method is the interest message being received. This parameter is an instance of a custom data structure used to represent an interest sent by the base station. By requiring the Application component to implement this interface, the architecture allows the application to have access to any new interest messages received by the routing layer. As mentioned above, this places the responsibility for determining whether or not an incoming interest is serviceable in the hands of the application layer.

4.2.6 Send Interface

The Send interface is a standard TinyOS interface and is used in the proposed routing architecture to allow the Application module to send messages via the implemented routing protocol. The structure of interface is as follows:

```
interface Send {
        command void* getBuffer(TOS_MsgPtr msg, uint16_t* length);
        command result_t send(TOS_MsgPtr msg, uint16_t length);
        event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

When the application wishes to send a message, it first declares an instance of a TOS_Msg (TinyOS Message), which represents a message to be sent over the radio. The format of a TinyOS message can be seen in Figure 21.

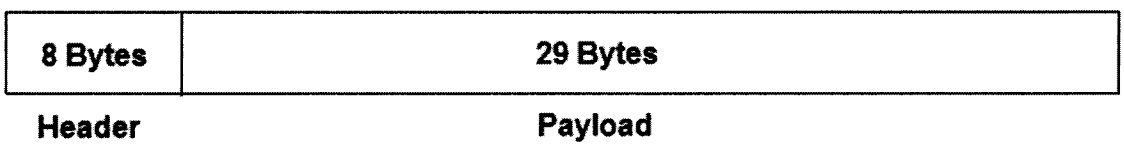


Figure 21 – TOS_Msg Format

Since both the application's data as well as the routing information must be stored in the packet payload, the application first calls the *getBuffer* method to get a pointer to the location in the payload where the data should be placed. The first parameter to this

method is the message to be sent while the second parameter is used by the method to notify the application of the maximum amount of space available for data.

Once the application has placed its data in the TOS_Msg, it calls the *send* method to transmit the message. The first parameter to this method is the message to be sent and the second parameter indicates the space used to store the data (in bytes).

Finally, after the message is sent, the Send interface triggers the *sendDone* event (implemented by the application) to indicate whether or not the transmission was successful.

Note that the Send interface does not allow the application to specify a destination address for the outgoing message. This is because it is assumed that all messages are being sent to a single destination: the base station.

4.2.7 SendMsg Interface

The SendMsg interface is used by the routing layer to transmit messages over the radio and is also a standard TinyOS interface. The structure of this interface is detailed below:

```
interface SendMsg
{
        command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg);
        event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

Similarly to the Send interface, messages are sent using the *send* method. The first parameter to this method is the destination node's address. The second parameter is used to indicate how much space is occupied by the data being sent (in bytes). Finally, the last parameter is a pointer to the actual message to be transmitted.

The main difference between this interface and the Send interface is that, when sending a message, a destination address can be specified. This makes the SendMsg interface well suited for use in the routing layer where defining the next hop for a message is a necessity.

All three implemented protocols make use of the SendMsg interface to send various types of messages. In Dynamic Source Routing, in addition to data messages, the SendMsg interface is used to send the Route Request, Route Reply, and Route Error messages. Similarly, Minimum Transmission Routing uses this interface to send both data messages and the periodic routing update messages. Directed Diffusion, however, requires no protocol-specific messages to be sent and uses the SendMsg interface only to send data messages. Also important is that the InterestManager component uses this interface to send interest and interest reinforcement messages.

4.2.8 ReceiveMsg Interface

The last interface used in the proposed software architecture is ReceiveMsg. This interface is also standard to TinyOS and is used by the Application, DataRouter, and InterestManager components to receive incoming messages. Below is the structure of this interface:

```
interface ReceiveMsg
{
        event TOS_MsgPtr receive(TOS_MsgPtr m);
}
```

The only member of this interface is the *receive* event, which takes a single TOS_Msg as a parameter (the incoming message). This event is triggered on reception of a message

and contains the logic necessary to deal with incoming messages of a specific type (i.e., for each type of message received by a module, there is a corresponding *receive* method).

In addition to receiving data messages, Dynamic Source Routing uses the ReceiveMsg to receive all protocol-specific messages such as Route Request, Route Reply, and Route Error messages. Likewise, in Minimum Transmission Routing, both data messages and routing update messages are received using the ReceiveMsg interface. Additionally, Directed Diffusion also uses this interface to access incoming data messages. Also of note is that the InterestManager component uses the ReceiveMsg interface to receive both interest and interest reinforcement messages.

4.3 Media Access Control

Also visible in Figure 20 is the GlobalMAC component, which provides a very simple implementation of a media access control protocol. All outgoing broadcast messages required in the software architecture are sent through the GlobalMAC component. Although this component is not a part of the core architecture, it is nonetheless an important module. This is because GlobalMAC is developed to solve a very common and classic computer networking predicament: the hidden terminal problem.

The hidden terminal problem occurs when two (or more) nodes not in transmission range of each other simultaneous attempt to transmit a message to a common neighbour, resulting in a corrupted message at the receiving node. This behavior is demonstrated in Figure 22.

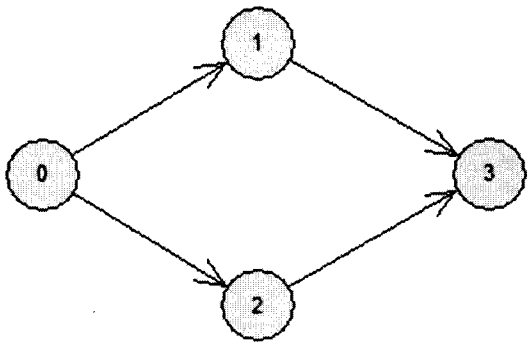


Figure 22 - Hidden Terminal Problem

In this scenario, assume that node 0 broadcasts an interest message that is received by nodes 1 and 2. Each of these nodes will attempt to re-broadcast the message at approximately the same time, resulting in a corrupted message being received by node 3. A positive acknowledgement (ACK) by the receiver (node 3) is usually a good defense against this problem, as the sender will receive confirmation that the message has been correctly received. In situations where messages are broadcasted, however, this approach is not sufficient (e.g., propagating interest messages).

The radio stack implementation provided with TinyOS provides media access control in the form of a Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) implementation. In CSMA/CA, the sending node listens to the communication medium to ensure that no one else is transmitting. If no radio activity is detected, the sending node will transmit its message. Although this approach works well for nodes within communication range, it does not solve the hidden terminal problem because the sending nodes cannot detect each other's radio activity.

In order to combat the hidden terminal problem and allow the experiments to be carried out, it is necessary to implement a network-wide media access control protocol. This implementation is contained in the GlobalMAC component. GlobalMAC solves the

problem described above by implementing a simple time division multiple access (TDMA) scheme for sending messages. Each node in the network is assigned a time slot in which it can send messages. If a node wishes to send a message, it must wait until it's next scheduled time slot in order to transmit. Although simplistic, this scheme guarantees that no two nodes in the network are broadcasting at the same time, thus eliminating the hidden terminal problem. Therefore, all messages broadcasted in the routing architecture are first sent to the GlobalMAC component after which they are sent over the radio.

It is important to note that, although this approach works well in the TOSSIM environment, it is not directly applicable to an actual sensor network deployment. This is because TOSSIM allows all nodes to have access to a global clock value that can be used to devise the TDMA scheme. In the real world, however, the clocks in each of the nodes are not synchronized, causing this time-sharing approach to fail. Additional measures would have to be taken to ensure time synchronization among all the nodes. An example of a clock synchronization algorithm for wireless sensor networks can be found in [29].

4.4 Experimentation Setup

In order to carry out the experiments, it is necessary to facilitate the collection of data from a simulated sensor network. To accomplish this, a Java program called BaseStation was developed that acts as a base station to inject interest messages into the network as well as process the data received from sensor nodes. A high level view of how the BaseStation program interacts with the simulated sensor network can be seen in Figure 23.

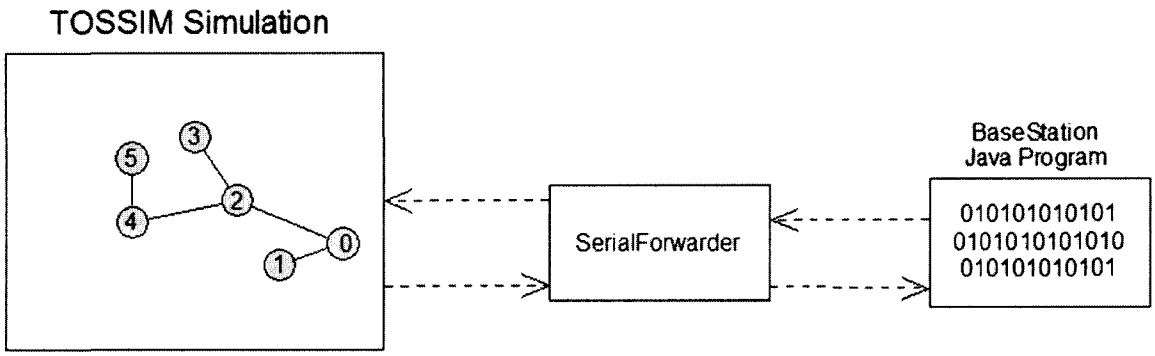


Figure 23 - Experimentation Setup

In order to communicate with the sensor network, the BaseStation program makes use of SerialForwarder. SerialForwarder, which is included in the standard TinyOS distribution, connects to node 0 in a running TOSSIM simulation over a virtual serial port [16]. During the simulation, any messages that node 0 sends over its serial port will be received by the SerialForwarder program (and vice versa). To allow external applications to communicate with the sensor network and receive data from the nodes, SerialForwarder also provides a TCP interface. This allows the BaseStation program to send and receive messages from the simulated sensor network.

As mentioned above, most TinyOS applications are written in nesC (a C-like language). Therefore, TinyOS packets received by the SerialForwarder program are stored in C data structures. In order to make it easy for Java client applications to interpret data received from the sensor network, a tool called the Message Interface Generator (MIG) is used to generate Java objects that correspond to TinyOS packet types [16]. When a TinyOS packet is received from SerialForwarder, the MIG tools will convert the data into a Java object that is then handed to the client application (e.g., the

BaseStation program). The Java object that is returned has all the appropriate getter and setter methods to allow the client application easy access to the sensor data.

To actually start the TOSSIM simulation with the desired parameters, the following command is executed:

./a.out -p -cpuprof -rf=cube6.nss -t=180 -b=1 36 > output.txt

Here, *a.out* represents the compiled TinyOS application (including a routing protocol implementation). The *p* argument indicates that the PowerTOSSIM plug-in should be used to calculate energy consumption statistics. Similarly, the *cpuprof* argument is used by the PowerTOSSIM plug-in to specify that CPU energy consumption should be included in these statistics. Additionally, the *rf=cube6.nss* argument indicates that a topology file with the name *cube6.nss* will be used to define the network topology used in the simulation. The *t=180* parameter instructs TOSSIM to run the simulation for 180 seconds (3 minutes) while the *b=1* parameter dictates that all nodes in the network must boot within the first second of the simulation. The *36* parameter indicates that 36 nodes are involved in the simulation. The final parameter, *output.txt,* simply defines the log file that the simulation output is placed in.

A small portion of the TOSSIM topology file used in the above command (i.e., cube6.nss) can be viewed in Figure 24. The format of each line in the topology file is as follows: the first token represents a source node, the second token represents a destination node, and the third token is a link error probability (between 0 and 1).

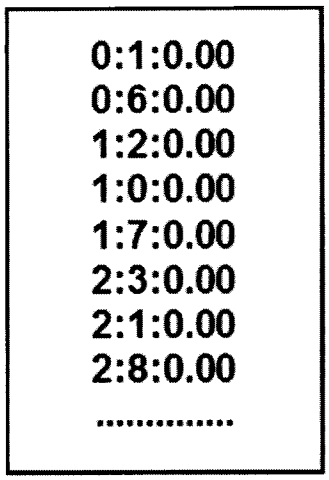


Figure 24 - Sample TOSSIM Topology File

For example, the first line in the above topology file dictates that node 0 has a communications link with node 1 and the error probability on that link is 0 (i.e., no bit errors can occur). Notice that the links denoted in the topology file are not bi-directional. If there exists a bi-directional link between two nodes, this must be representing using two entries in the topology file. A complete version of the above topology file can be found in Appendix B: TOSSIM Grid Topology File.

4.5 Summary

In this chapter, a software architecture to implement routing protocols for sensor networks is proposed. The individual components of this architecture are described as well as the interfaces that provide communication between these components. Each of the three selected routing protocols is implemented in this architecture to allow for the experimentations to be run. Additionally, the need for media access control is explained along with a description of the GlobalMAC program module. Finally, the experimentation setup is discussed, showing how various software packages are used to

carry out the necessary experiments. In the next chapter, the results of these experiments are presented.

Chapter 5 – Experimentation Results

5.1 Overview

In this chapter, the results of performing the experimentations detailed previously are provided. For each scenario, the results are first presented and then analyzed in order to explain the outcomes. The results for static metrics are detailed first, followed by the results and analysis of the Stabilization Phase in Minimum Transmission Routing. The Perfect Scenario results are described next, followed by the Error Scenario outcomes. For both the Perfect and Error Scenarios, results for all metrics are listed first for the grid topology and then for the random topology.

5.2 Static Metrics

This section details the results for static metrics, or metrics for which the results are the same among all scenarios and networking topologies. Code Size is the only example of such a metric in the experimentations.

As mentioned in Chapter 3, knowing how much program and data memory is occupied by each routing protocol implementation is very important. If the routing protocol occupies a large amount of memory, there are fewer resources available to the actual application collecting the sensor data. The results for the Code Size metric can be seen in Figure 25. The routing protocols are listed along the horizontal axis and the Code Size (in bytes) is displayed along the vertical axis. Also notice that the value for the Code Size is divided into both the Program Code Size and Data Size components.

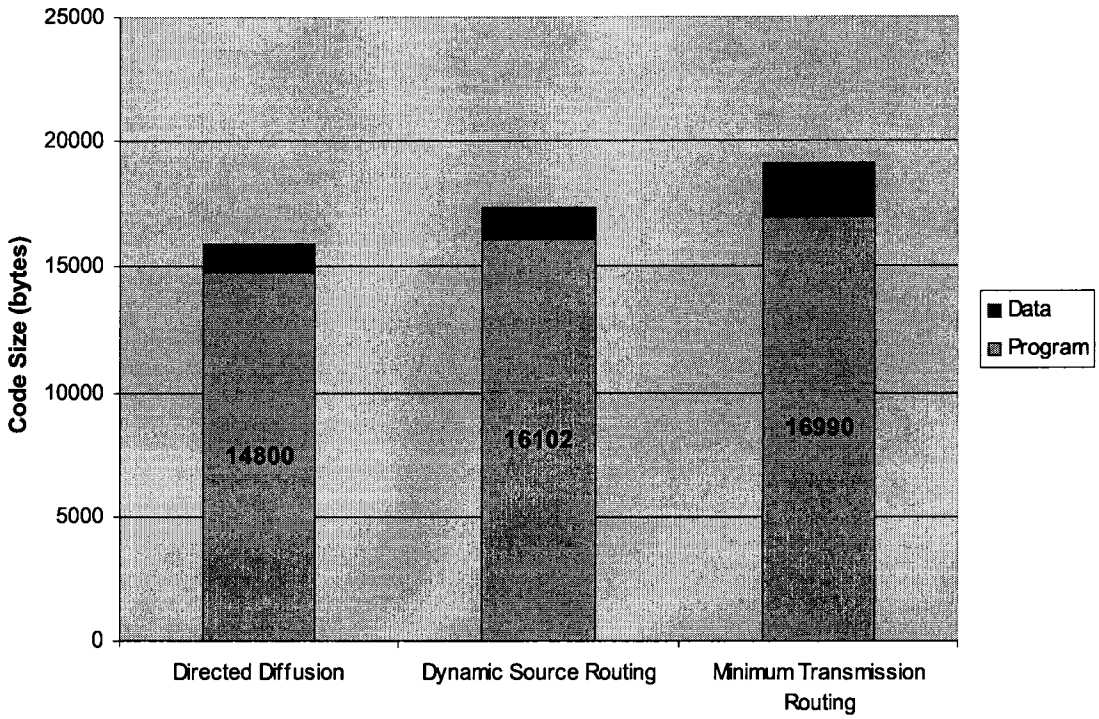


Figure 25 - Code Size Results

As is evident in the above figure, Directed Diffusion achieves the smallest Code Size with a value of 15929 bytes. Dynamic Source Routing attains a slightly higher value of 17368 bytes. Finally, Minimum Transmission Routing scores a value of 19159 bytes.

Additionally, provided below in Figure 26 is a chart depicting the approximate number of lines of source code (non-commented) for each routing protocol implementation. Dynamic Source Routing and Minimum Transmission Routing contain more lines of code, weighing in at 1000 and 1100 lines (respectively). Notice that Directed Diffusion achieves a significantly smaller number at 450 lines of code. Also of note is that the InterestManager component (not shown in Figure 26) contains approximately 650 lines of code and is included in the routing layer no matter which protocol is used.

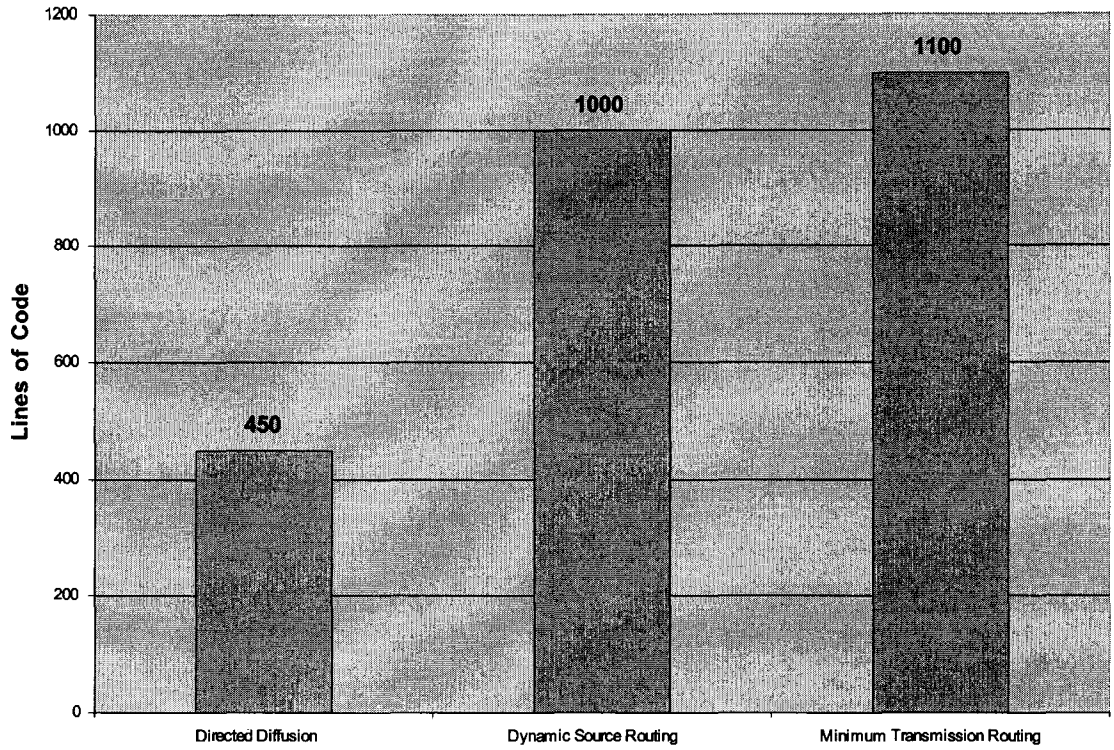


Figure 26 - Code Size (in lines of code)

The fact that Directed Diffusion has the smallest Code Size stems from its use of the interest cache as the basis for making routing decisions. As noted in the description of the software architecture, the InterestManager component (which contains the interest cache) is common to all three protocols. Therefore, the Directed Diffusion protocol does not require additional storage tables such as the route cache in Dynamic Source Routing or the neighbour tables in Minimum Transmission Routing, allowing it to achieve the smallest Code Size. Also, since the logic to deal with interests in encapsulated within the InterestManager component, this minimizes the amount of program memory needed for the Directed Diffusion protocol to function properly.

5.3 Minimum Transmission Routing (Stabilization Phase)

For the reasons discussed in Chapter 3, the Stabilization Phase in Minimum Transmission Routing is analyzed separately. There are only three metrics that are relevant to the Stabilization Phase: Total Energy Consumption, Total Number of Messages Transmitted, and Energy Consumption Variation. The results for each of these metrics are presented below for both the grid and random topologies under the Perfect Scenario. The Error Scenarios are not considered because, as previously stated, Minimum Transmission Routing is not relevant in these scenarios.

5.3.1 Total Energy Consumption

The first metric considered for the Stabilization Phase is the Total Energy Consumption. The results for this metric can be viewed in Figure 27. The topologies are listed along the horizontal axis while the Total Energy Consumption is shown along the vertical axis (in mJ).

In the grid topology, the Minimum Transmission Routing Stabilization Phase results in Total Energy Consumption of 161.18 mJ. Similarly, in the random topology, the value for the Total Energy Consumption is 117.02 mJ.

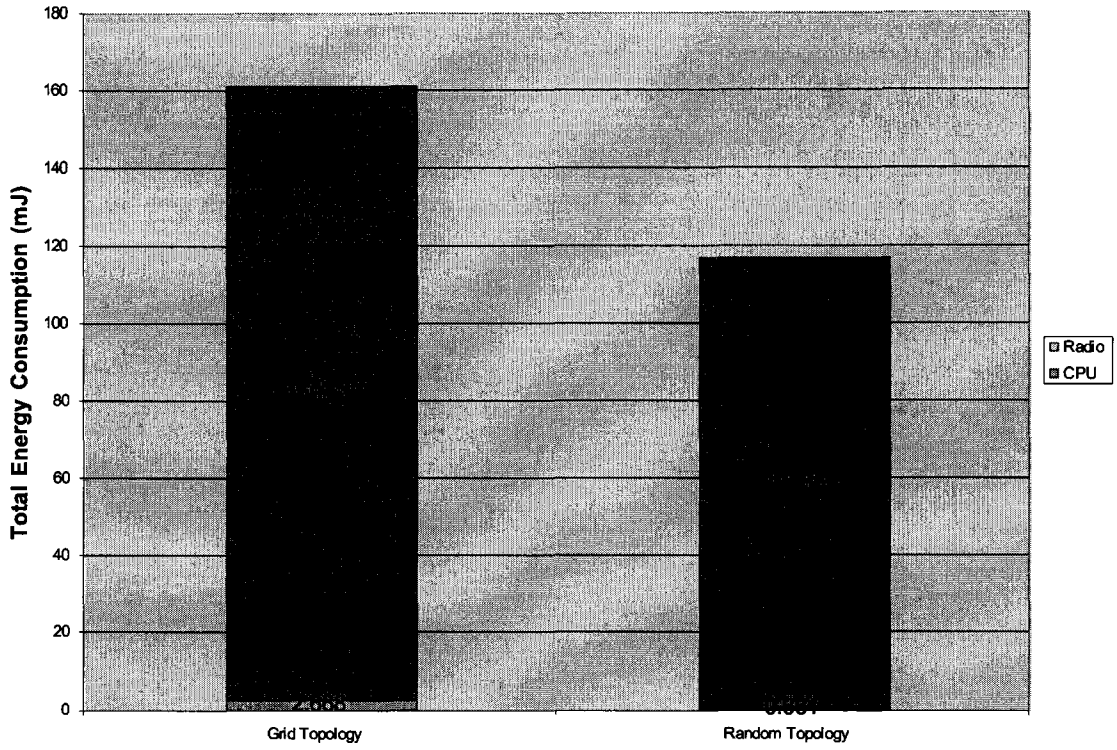


Figure 27 – Total Energy Consumption (Stabilization Phase)

Note that the energy consumed by the radio accounts for the vast majority of the Total Energy Consumption in both topologies. As well, the Stabilization Phase appears to require a lower amount of energy in the Random Topology as opposed to the Grid Topology.

5.3.2 Total Number of Messages Transmitted

The second metric considered for the Stabilization Phase of Minimum Transmission Routing is the Total Number of Messages Transmitted. The outcomes for this metric are presented in Figure 28. Again, the specific network topologies are shown along the horizontal axis while the messages transmitted are displayed along the vertical axis.

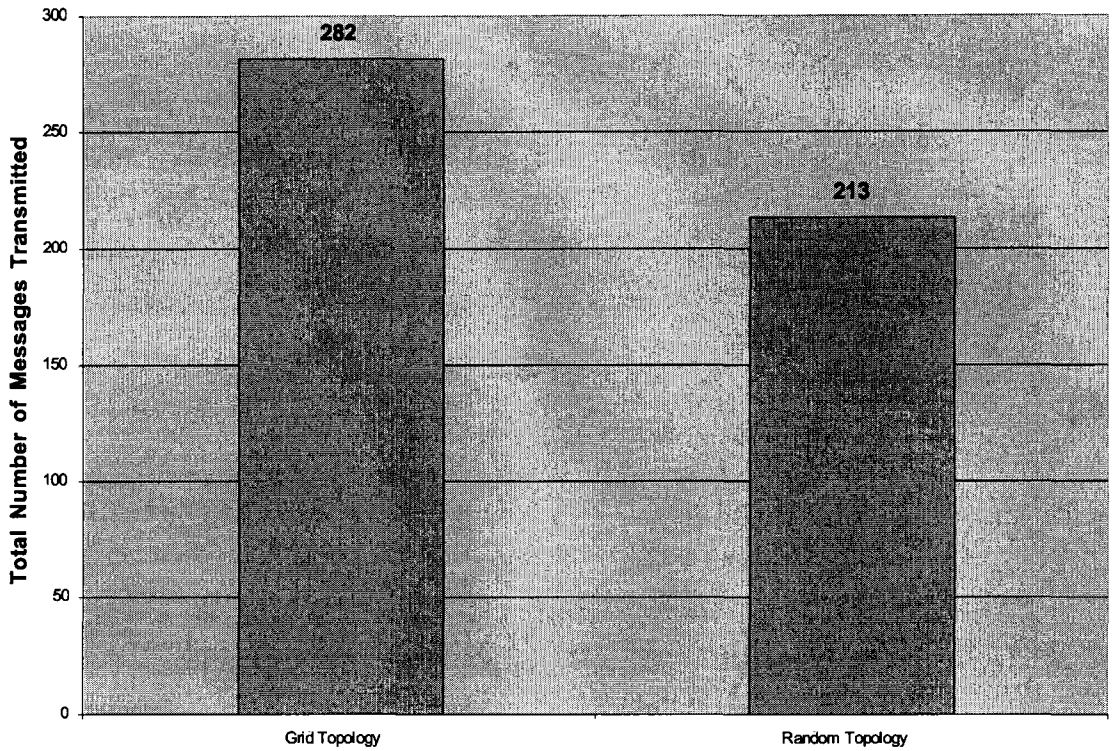


Figure 28 - Total Number of Messages Transmitted (Stabilization Phase)

In the grid topology, a Total Number of Messages Transmitted value of 282 is achieved. The value obtained in the random topology is a little less at 213 messages. Notice these values correspond to the ones presented for the Average Dissipated Energy metric. For instance, the random topology requires fewer messages be sent and therefore achieves a lower value for the Average Dissipated Energy.

5.3.3 Energy Consumption Variation

The third, and final metric analyzed for the Stabilization Phase is the Energy Consumption Variation. The results for this metric are seen in Figure 29 with the computed Energy Consumption Variation shown along the vertical axis (in mJ).

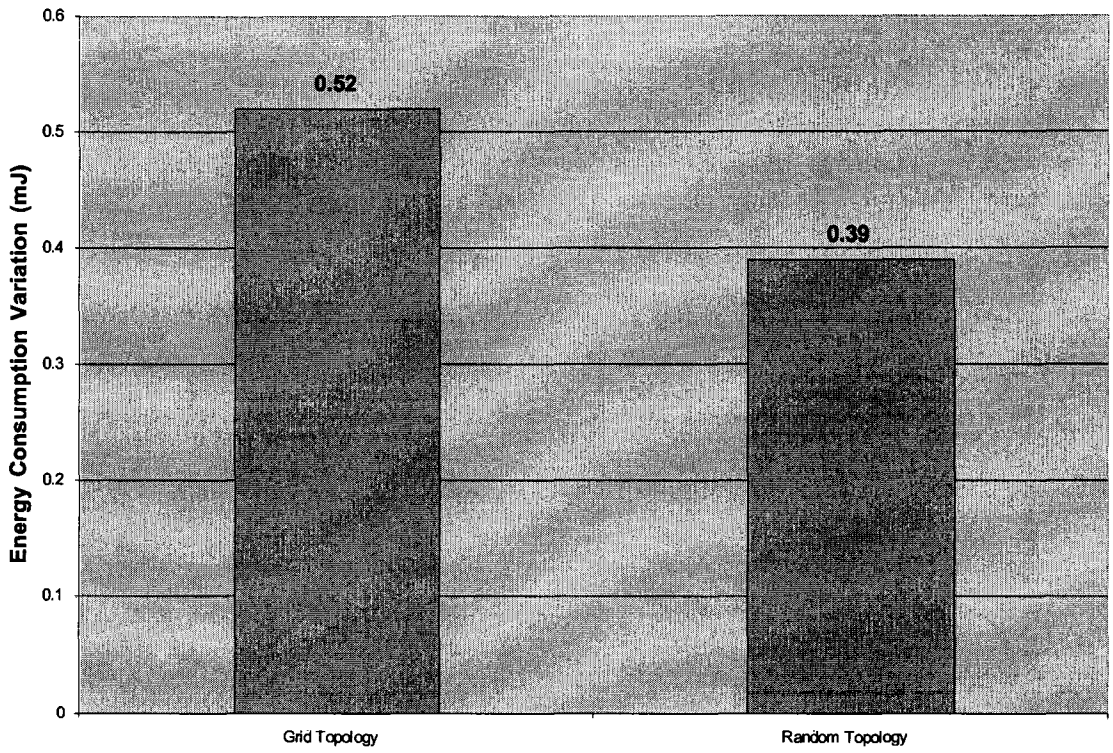


Figure 29 - Energy Consumption Variation (Stabilization Phase)

For the grid topology, an Energy Consumption Variation value of 0.52 mJ is achieved. In the random topology, a lower value of 0.39 mJ is measured. These values are reasonably small, suggesting that all nodes in the network consume a similar amount of energy during the Stabilization Phase. This result is not surprising since, during the Stabilization Phase, only route update messages are being sent throughout the network. Since each node in the network periodically sends these messages, it is expected that the value for

the Energy Consumption Variation will be small. Also, the random topology achieves a slightly better value compared to the grid topology due to the larger number of potential (and therefore unutilized) routes available in the grid topology.

5.4 Perfect Scenario Grid Topology Results

In this section, the experimentation results for each metric under the perfect scenarios, using the grid topology, are presented.

5.4.1 Average Dissipated Energy

The first metric for which results are presented is Average Dissipated Energy. As mentioned previously, this is perhaps the most important metric because energy consumption is directly linked to the lifetime of the sensor network. A graphical representation of the results for this metric can be seen in Figure 30. In this chart, the horizontal axis denotes specific routing protocols and the vertical axis is the Average Dissipated Energy (in mJ per message). Also note that the value for each routing protocol is divided into the Average Dissipated Energy for both the CPU and radio.

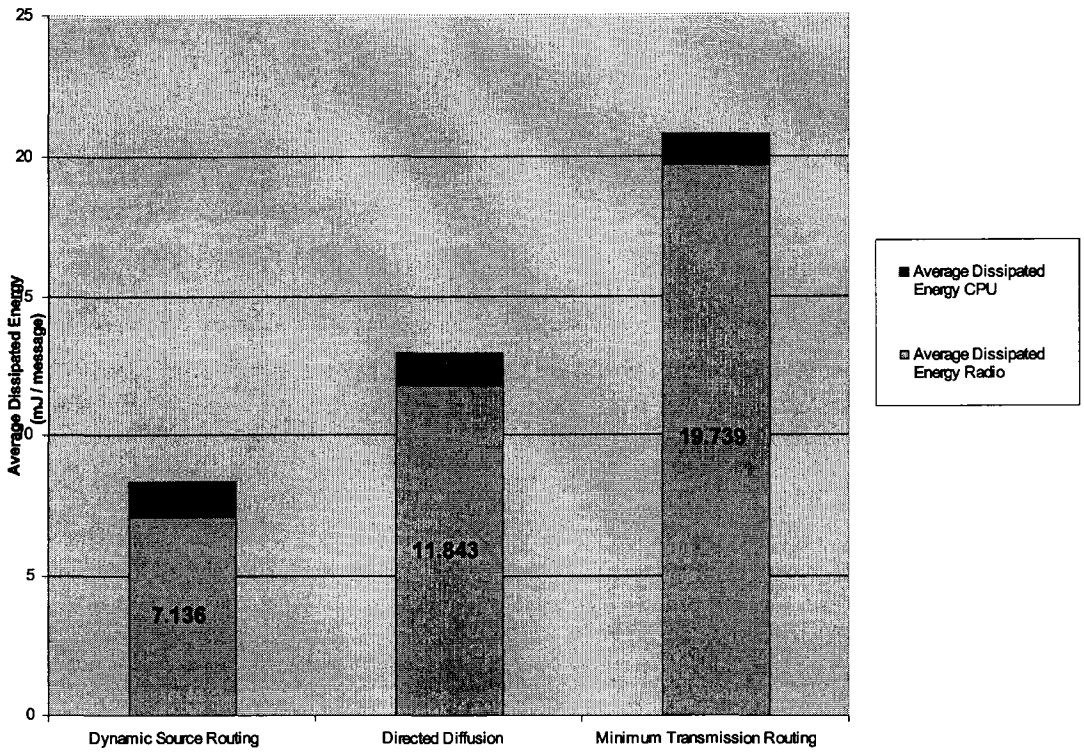


Figure 30 - Average Dissipated Energy (Perfect Scenario, Grid Topology)

As is visible in the above figure, Dynamic Source Routing achieves the lowest Average Dissipated Energy at a value of 8.4 mJ. Directed Diffusion has a moderately higher value of 13 mJ followed by Minimum Transmission Routing with a value of 20.8 mJ. Also notice that the energy consumed by the CPU for each of the routing protocols is relatively the same. This shows that the Average Dissipated Energy is primarily determined by the energy utilized for the radio to send messages.

The fact that Dynamic Source Routing achieves the best value for Average Dissipated Energy is not surprising in the perfect scenarios. Since no errors occur, route discovery is only performed once and a single route is used throughout the entire experiment to transmit data back to the base station. In Directed Diffusion, however, the

multiple routes being used to transmit data during the initial interest period result in a higher volume of messages transmitted and therefore a higher Average Dissipated Energy value. In Minimum Transmission Routing, route update messages are sent periodically for the entire experiment, resulting in the highest value for Average Dissipated Energy. It is important to note, however, that only one route is constructed in the Dynamic Source Routing protocol while all routes are inherently constructed in Minimum Transmission Routing. That is, in Minimum Transmission Routing, all nodes are able to send data back to the base station.

5.4.2 Average Delay

The next metric considered is Average Delay, or the average time required for data to travel from the data generating node to the base station. The results for this metric can be viewed in Figure 31.

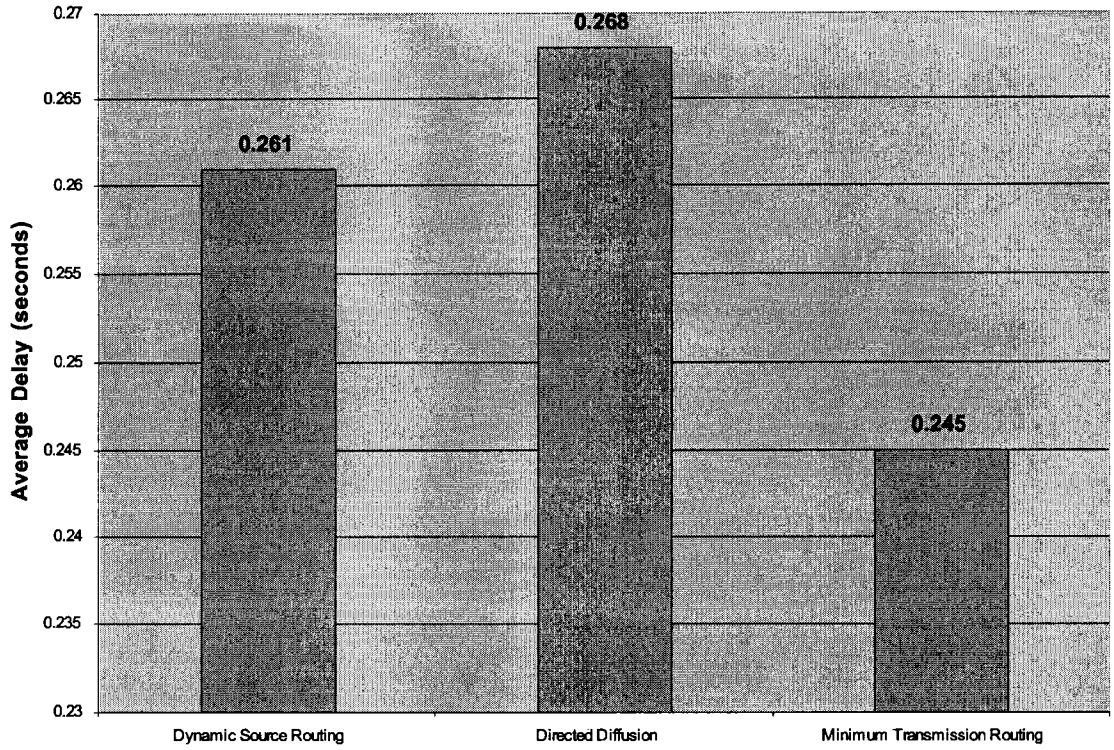


Figure 31 - Average Delay (Perfect Scenario, Grid Topology)

Again, the routing protocols are listed along the horizontal axis with their corresponding Average Delay value (in seconds) on the vertical axis. All three protocols achieve very similar results for Average Delay with Minimum Transmission Routing having a slightly superior value of 0.245 s. Dynamic Source Routing and Directed Diffusion follow close behind with values of 0.261 s and 0.268 s respectively.

These results indicate that the number of hops required for data to travel from the data generator to the base station is very comparable for all three routing protocols, resulting in nearly identical message transmission times.

5.4.3 Energy Consumption Variation

Results for Energy Consumption Variation, determined by calculating the standard deviation over the energy consumption for all sensor nodes, are presented next.

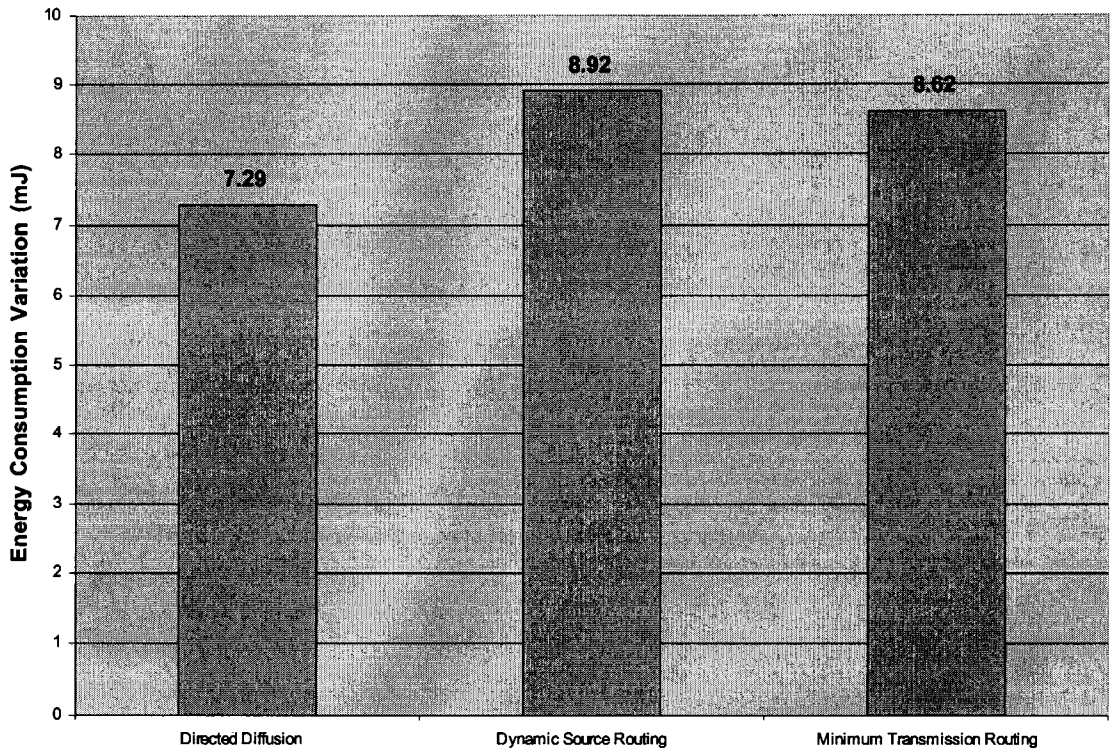


Figure 32 - Energy Consumption Variation (Perfect Scenario, Grid Topology)

The results for this metric are displayed in Figure 32. Here, the performance of each routing protocol varies. Directed Diffusion fares best with a value of 7.29 mJ. Minimum Transmission Routing and Dynamic Source Routing achieve higher values of 8.62 mJ and 8.92 mJ, respectively.

The slightly higher values for Dynamic Source Routing and Minimum Transmission Routing are caused by the fact that both of these protocols use a single path to transmit data for the entirety of the experiment. The result is greater energy

consumption in nodes along this path, thereby increasing the energy consumption variation amongst all nodes in the network.

5.4.4 Total Number of Messages Transmitted

The last metric considered under this set of experiments is the Total Number of Messages Transmitted. The results are shown in Figure 33 with the Total Number of Messages Transmitted indicated on the vertical axis.

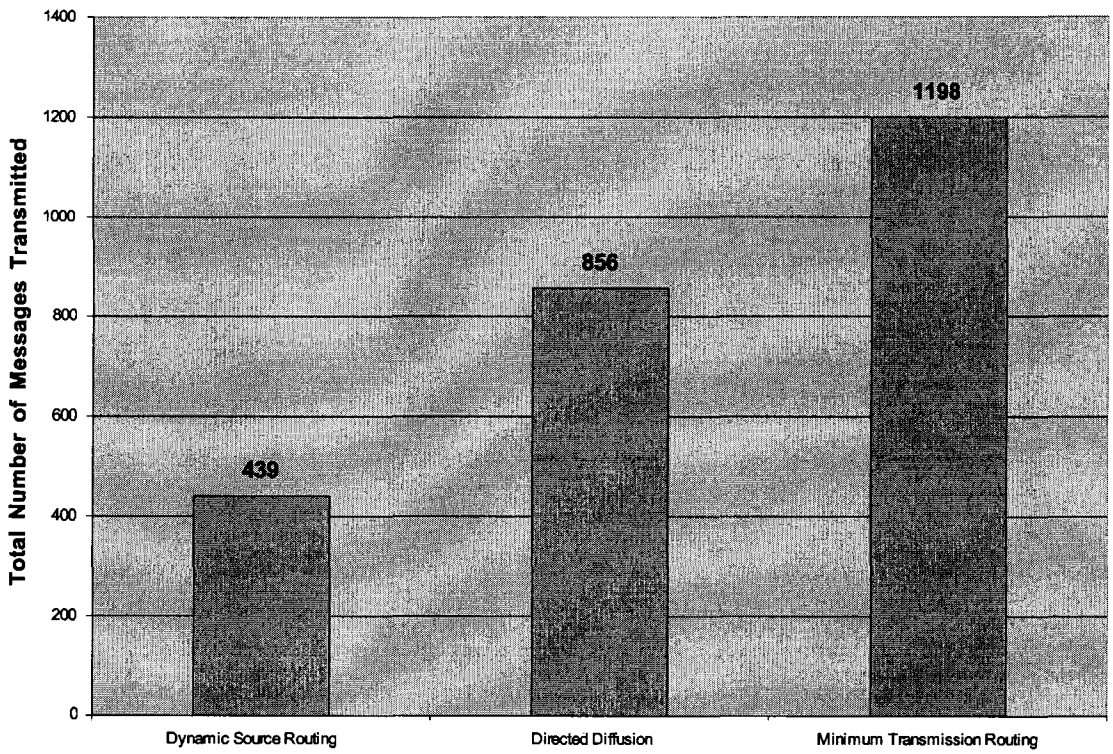


Figure 33 - Total Number of Messages Transmitted (Perfect Scenario, Grid Topology)

Dynamic Source Routing obtains the lowest number of messages transmitted with a value of 429. Directed Diffusion fares worse, obtaining a value of 856 messages, followed by Minimum Transmission Routing which sends 1198 messages.

Notice that these results correspond very closely to those for the Average Dissipated Energy metric. This behavior is expected because, as mentioned above, the radio is the dominant source of energy consumption in these experiments. Therefore, it follows that the protocol that has the lowest Average Dissipated Energy will also have the lowest value for the Total Number of Messages Transmitted. As a result, Dynamic Source Routing performs best in this metric, followed by Directed Diffusion and Minimum Transmission Routing.

5.5 Perfect Scenario Random Topology Results

In this section, the experimentation results for each metric under the perfect scenarios using the random topology will be presented.

5.5.1 Average Dissipated Energy

The results for the Average Dissipated Energy metric under the random topology are displayed in Figure 34.

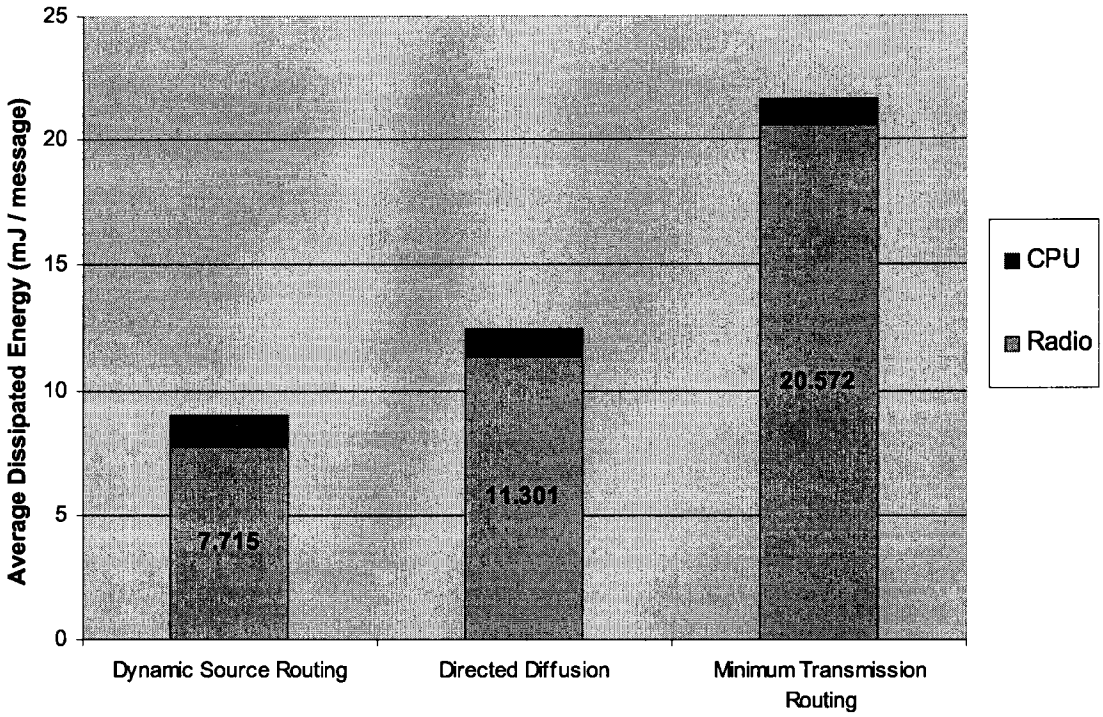


Figure 34 - Average Dissipated Energy (Perfect Scenario, Random Topology)

Dynamic Source Routing again achieves the lowest Average Dissipated Energy value of 8.99 mJ. Directed Diffusion obtains a slightly higher Average Dissipated Energy with a value of 12.48 mJ. Finally, Minimum Transmission Routing scores the worst value for Average Dissipated Energy at 21.47 mJ.

These results are nearly identical to those presented for the grid topology. Dynamic Source Routing has a slightly worse Average Dissipated Energy than in the grid topology, and both Directed Diffusion and Minimum Transmission Routing perform marginally better. These variations are not significant, however, indicating that changing the network topology has little effect on the performance of the routing protocols.

5.5.2 Average Delay

Next, the results for the Average Delay metric are presented in Figure 35.

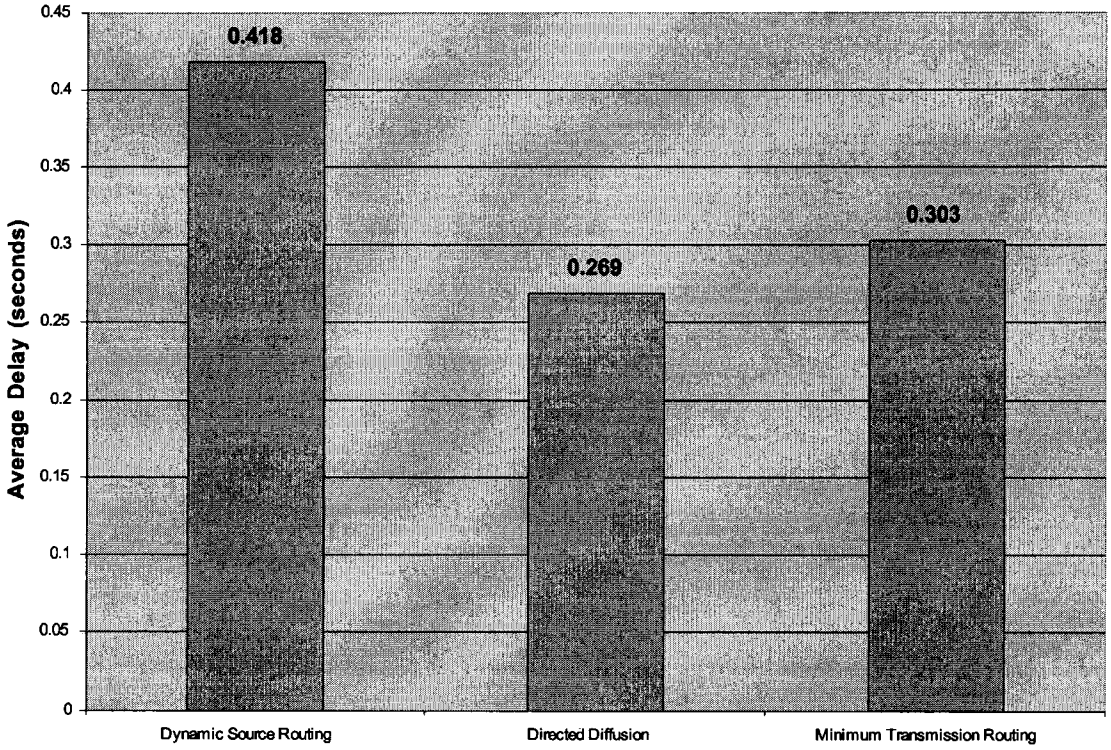


Figure 35 - Average Delay (Perfect Scenario, Random Topology)

Again, the results are very similar to those under the grid topology. Directed Diffusion performs best this time, achieving an Average Delay of 0.269 s. Minimum Transmission Routing scores slightly higher with an Average Delay of 0.303 s and Dynamic Source Routing attains a slightly worse Average Delay value of 0.418 s.

As mentioned above, the fact that the values for Average Delay are comparable indicates that the data path used by each protocol contains roughly the same number of hops. The other factor that causes a small amount of variation is the number of message retransmissions at the data link layer due to no acknowledgement message being received by the sending node.

5.5.3 Energy Consumption Variation

The results for the Energy Consumption Variation metric are displayed in Figure 36.

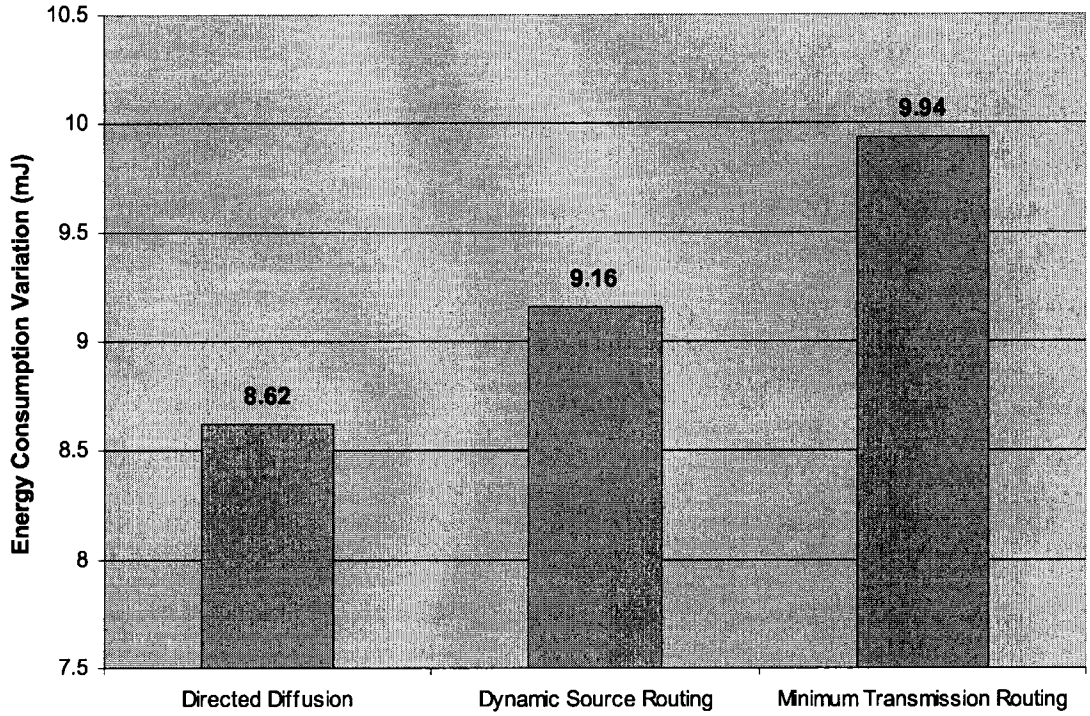


Figure 36 - Energy Consumption Variation (Perfect Scenario, Random Topology)

As with the grid topology results, Directed Diffusion again has the lowest Energy Consumption Variation with a value of 8.62 mJ. Dynamic Source Routing obtains a measurement of 9.16 mJ, followed by Minimum Transmission Routing with a value of 9.94 mJ.

Directed Diffusion achieves the best performance in this experiment for the same reasons indicated in the grid topology results: the other protocols use a single data path for the entire simulation, resulting in a higher Energy Consumption Variation.

5.5.4 Total Number of Messages Transmitted

The outcome for the Total Number of Messages Transmitted metric is revealed in Figure 37.

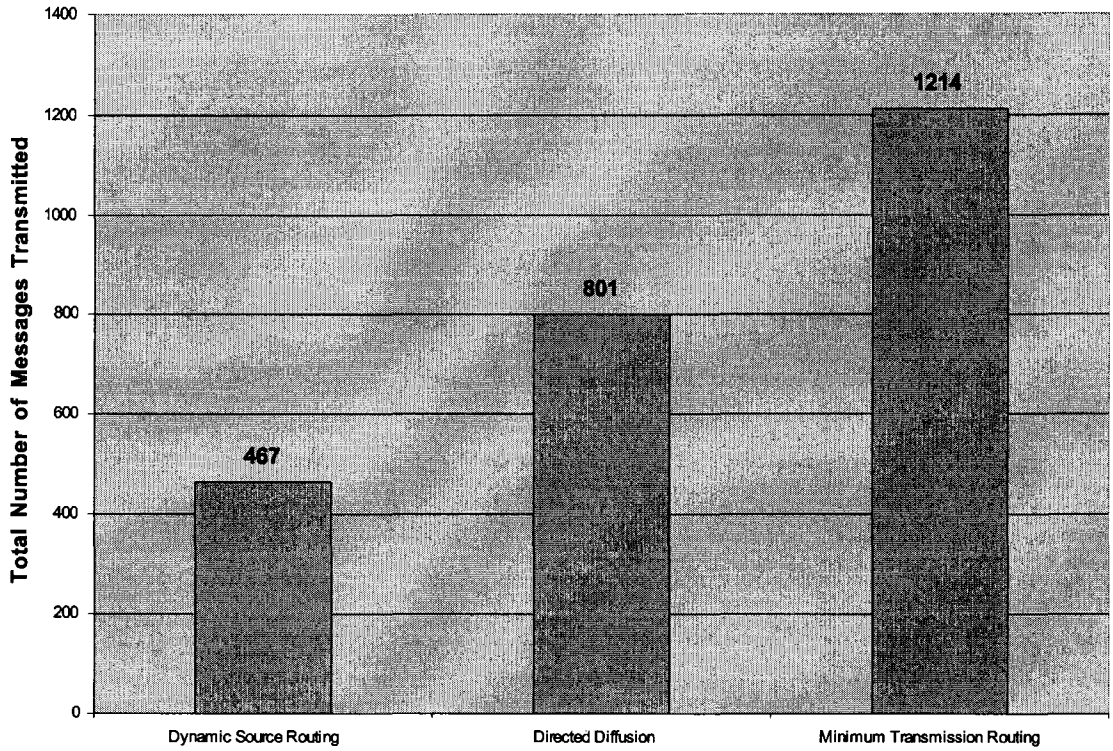


Figure 37 - Total Number of Messages Transmitted (Perfect Scenario, Random Topology)

Once again, Dynamic Source Routing has the lowest number of messages transmitted with a value of 467. Directed Diffusion fares worse off with 801 messages transmitted while Minimum Transmission Routing requires a substantially higher number of messages with a value of 1214.

Again, these results directly correlate to those for the Average Dissipated Energy metric based on the logic explained previously. As well, adding to a noticeable trend, the results are also very close to those for the grid topology, indicating that each protocol sends roughly the same amount of messages regardless of which topology is used.

5.6 Error Scenario Grid Topology Results

In this section, the experimentation results for each metric under the error scenarios using the grid topology are presented.

5.6.1 Recovery Total Energy Consumption

First of all, the results for the Recovery Total Energy Consumption metric, or the total amount of energy required to recover from a network error, are presented. These results are displayed in Figure 38. Along the horizontal axis, the results for both Dynamic Source Routing and Directed Diffusion are shown for each of the three error scenarios (closest, intermediate, and farthest). The Total Energy Consumed (in mJ) is shown along the vertical axis. Notice that each value is divided into both the energy consumed by the radio and the energy consumed by the CPU.

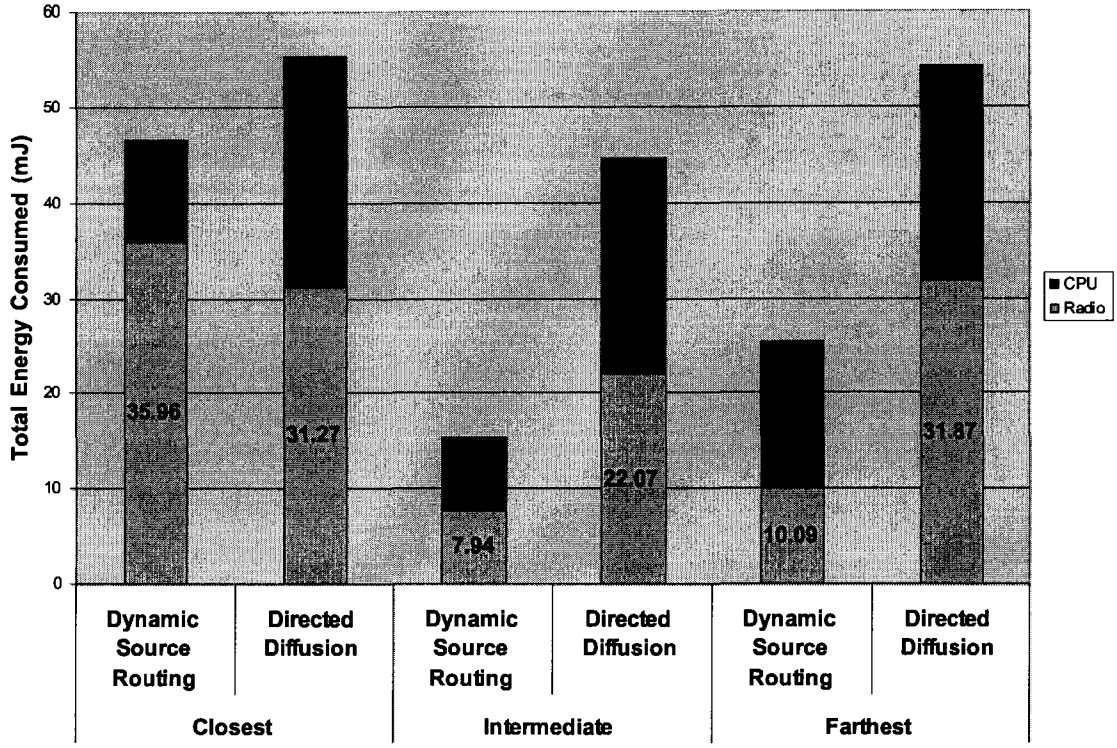


Figure 38 - Recovery Total Energy Consumption (Error Scenario, Grid Topology)

In the closest error scenario, Dynamic Source Routing achieves a Recovery Total Energy Consumption value of 46.48 mJ while Directed Diffusion fares slightly worse off with a value of 55.37 mJ. These results are similar because, in this experiment, both protocols required a message to be flooded throughout the network for stabilization to occur. In the case of Directed Diffusion, a new interest message must always be broadcasted through the network when an error is detected. Dynamic Source Routing, on the other hand, only requires flooding route request messages when an error is detected and no other cached route to the base station exists. In this case, the data generating node *did not* have another cached route, resulting in the flooding of a new route request message.

Dynamic Source Routing also performed better in the intermediate error scenario, obtaining a far superior Recovery Total Energy Consumption value of 15.42 mJ compared to the Directed Diffusion value of 44.67 mJ. As mentioned above, Directed Diffusion must broadcast a new interest message when an error is detected. In this experiment, however, the data generating node in the Dynamic Source Routing protocol did have another cached route to the base station when the network error was detected. This avoided having to broadcast another Route Request message and leads to a much more efficient Recovery Total Energy Consumption value for Dynamic Source Routing.

Finally, in the farthest error scenario, Dynamic Source Routing once again has the advantage with a Recovery Total Energy Consumption value of 25.53 mJ as opposed to the Directed Diffusion value of 54.26 mJ. Once again, the data generating node in the Dynamic Source Routing protocol was able to utilize another stored route to the base station, resulting in a huge performance gain. Notice, however, that Dynamic Source Routing does consume more energy in this error scenario than it does in the intermediate scenario. This is because the network failure occurs farther away from the data generating node, resulting in the route error message having to travel a greater number of hops. The increased number of transmissions for the route error message thus accounts for the higher Recovery Total Energy Consumption value in the farthest error scenario.

Notice that the energy consumed by the radio and CPU is much more comparable than in the perfect scenarios. This is due to the fact that, during the error scenarios, each node is sending only a small number of messages. Thus, the radio accounts for a much smaller portion of the energy consumed than previously seen.

5.6.2 Total Number of Messages Transmitted

The results for the Total Number of Messages Transmitted metric are presented next and can be viewed in Figure 39.

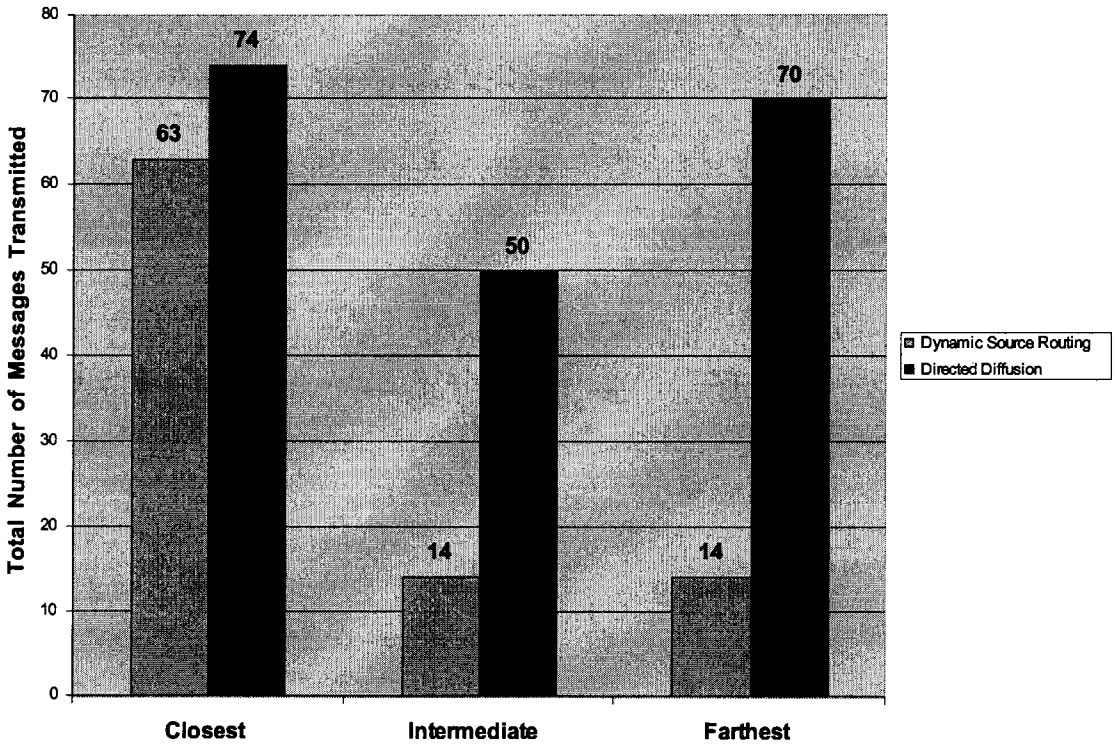


Figure 39 - Total Number of Messages Transmitted (Error Scenario, Grid Topology)

In the closest error scenario, both protocols achieve similar values with Dynamic Source Routing at 63 messages and Directed Diffusion at 74 messages. As described above, this similarity is due to both protocols having to rebroadcast a message to achieve network stabilization. Directed Diffusion does transmit a slightly greater number of messages, however, because of the initial exploratory interest sent after the error is detected. This interest message results in data flowing back to the base station along multiple paths,

accounting for the higher metric value. In contrast, Dynamic Source Routing routes data to the base station using only a single path.

For the intermediate scenario, Dynamic Source Routing obtains a superior result, requiring only 14 messages to be transmitted in order for the network to stabilize. Directed Diffusion, on the other hand, requires 50 messages to achieve the same result. In this experiment, the large difference is due to the fact that, when the network error occurs, Dynamic Source Routing has another cached route to the base station. This eliminates the need to broadcast another route request message and minimizes the number of messages transmitted.

Finally, in the farthest scenario, Dynamic Source Routing again achieves an efficient result of 14 messages, compared to 70 messages for Directed Diffusion. Similarly to the intermediate error scenario, Dynamic Source Routing is able to use another cached route to the base station to avoid flooding messages throughout the network.

5.6.3 Number of Data Messages Lost

Next, the experiment outcomes for the Number of Data Messages Lost metric are discussed. As previously mentioned, it is essential for a routing protocol to minimize lost data messages during a network failure. With this in mind, the results for each routing protocol under all three error scenarios are presented in Table 4.

	Closest	Intermediate	Farthest
Dynamic Source Routing	0	0	0
Directed Diffusion	1	1	1

Table 4 - Number of Data Messages Lost (Error Scenario, Grid Topology)

Perhaps the most noticeable aspect of these results is that they are consistent over the closest, intermediate, and farthest error scenarios. In other words, the placement of the network error does not affect the behavior of the routing protocol. As mentioned in Chapter 2, Dynamic Source Routing is able to explicitly detect network errors and the data generating node can cache outgoing data messages until a new route to the base station is discovered. This ability to cache messages (albeit a limited number) allows Dynamic Source Routing to avoid losing any data messages during each of the error scenarios.

Directed Diffusion, on the other hand, has no such ability to instantly detect network errors and does not cache data messages. Thus, the network failure is not detected until the base station realizes that it has not received the next data message as expected. This results in the loss of a single data message across all error scenarios. On a positive note, Directed Diffusion is able to stabilize quickly enough to prevent the loss of further data messages.

5.6.4 Stabilization Time

The final metric considered is Stabilization Time. Figure 40 contains the results for this metric.

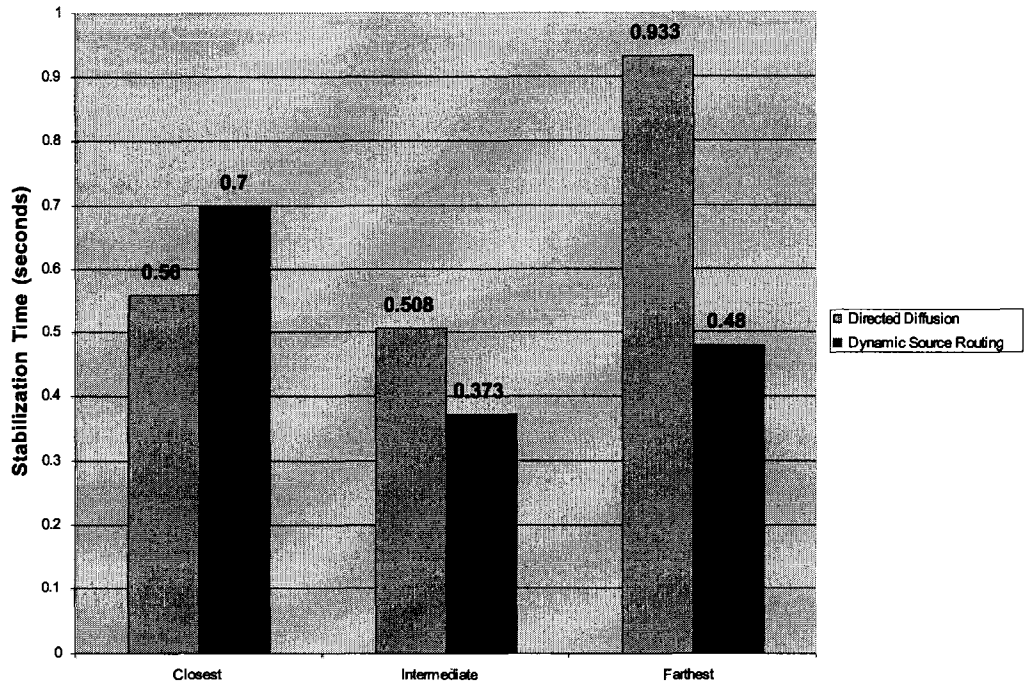


Figure 40 - Stabilization Time (Error Scenario, Grid Topology)

In the closest error scenario, Directed Diffusion attains the lowest Stabilization Time with a value of 0.56s while Dynamic Source Routing scores slightly higher at 0.7 s. In the intermediate scenario, however, Dynamic Source Routing achieves the better result with a Stabilization Time of 0.373 s followed by Directed Diffusion with a value of 0.508. Likewise, in the farthest error scenario, Dynamic Source Routing obtains network stabilization in 0.48 s as opposed to the 0.933 s required by Directed Diffusion.

In the closest error scenario, as mentioned during the discussion for the Recovery Total Energy Consumption metric, both routing protocols must perform flooding to achieve stabilization. Here, Directed Diffusion scores slightly better than Dynamic Source Routing due to the sequence of events that take place in both protocols after a network error is detected. Both protocols must broadcast a message (either an interest or a route request) throughout the network before data can be routed to the base station. In

Dynamic Source Routing, however, the route error message must propagate back to the data generating node before the route request can be broadcasted. This introduces an additional delay to Dynamic Source Routing that is not seen with Directed Diffusion.

In the intermediate and farthest error scenarios, however, Dynamic Source Routing has another cached route to the base station when the network error is detected. This eliminates the need to flood a route request and allows data to be properly routed without any interruption. Directed Diffusion, on the other hand, must flood an interest message in these scenarios. This accounts for the improved performance of Dynamic Source Routing in the intermediate and farthest error scenarios.

5.7 Error Scenario Random Topology Results

In this section, the experimentation results for each metric under the error scenarios using the random topology are be presented.

5.7.1 Recovery Total Energy Consumption

As previously done, the experiment outcomes for the Recovery Total Energy Consumption metric are considered first. These results are shown below in Figure 41. The routing protocols are listed along the horizontal axis while the values for Recovery Total Energy Consumption are shown along the vertical axis.

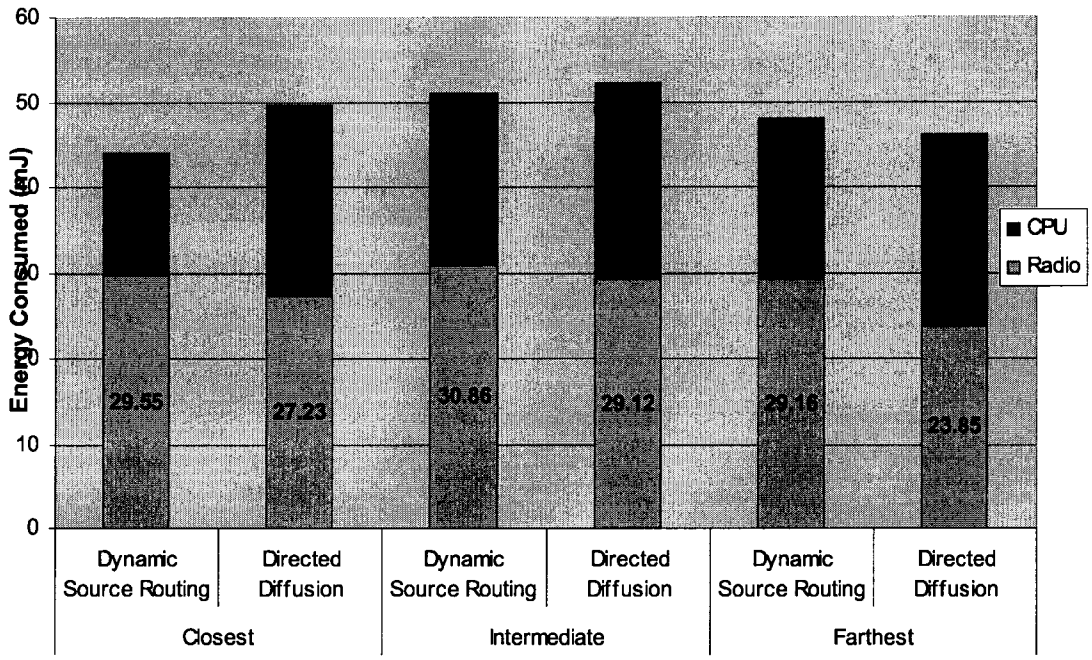


Figure 41 - Recovery Total Energy Consumption (Error Scenario, Random Topology)

In the closest error scenario, Dynamic Source Routing obtains a lower Recovery Total Energy Consumption of 43.98 mJ. The value for Directed Diffusion is moderately higher at 49.65 mJ. Notice that, although Dynamic Source Routing expends more radio energy than Directed Diffusion, it uses much less CPU energy, resulting in a better value for this metric.

Similar outcomes are evident in the intermediate error scenario. Again, Dynamic Source Routing achieves the lowest Recovery Total Energy Consumption with 51.12 mJ. Directed Diffusion follows very closely behind with a value of 52.17 mJ. Also of interest is the increase in CPU energy for Dynamic Source Routing compared with the closest error scenario.

The results for the farthest error scenario are somewhat different, however. In this case, Directed Diffusion obtains the best Recovery Total Energy Consumption with a value of 46.04 mJ. Dynamic Source Routing achieves a slightly higher value of 48.06 mJ. The better showing for Directed Diffusion is caused by a decrease in radio energy consumed as compared to the previous error scenarios.

Again notice that the CPU and radio energy consumption values are comparable due to the reasons outlined in section 5.6.1.

5.7.2 Total Number of Messages Transmitted

Next, the results for the Total Number of Messages Transmitted metric are discussed. These results can be viewed in Figure 42.

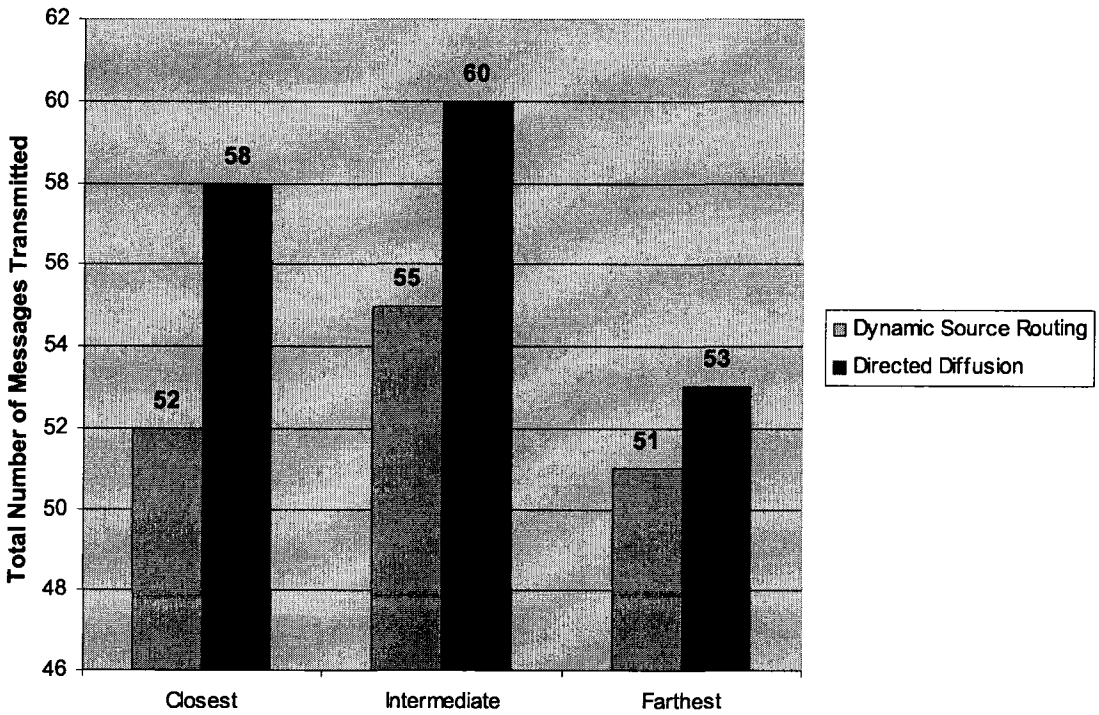


Figure 42 - Total Number of Messages Transmitted (Error Scenario, Random Topology)

In the closest error scenario, Dynamic Source Routing requires that 52 messages be sent to stabilize the network as opposed to the 58 required by Directed Diffusion. Similarly, in the intermediate error scenario, Dynamic Source Routing achieves stabilization using 55 messages while Directed Diffusion utilizes 60. Finally, in the farthest error scenario, network stabilization in the Dynamic Source Routing needs 51 messages while Directed Diffusion requires 53.

The results for all three error scenarios are consistent, with Dynamic Source Routing always edging out Directed Diffusion by requiring a slightly smaller number of messages. As mentioned before, this is due to the multiple data paths being used in Directed Diffusion in response to the initial exploratory interest.

5.7.3 Number of Data Messages Lost

The results for the Number of Data Messages Lost are revealed in Table 5.

	Closest	Intermediate	Farthest
Dynamic Source Routing	0	0	0
Directed Diffusion	1	1	1

Table 5- Number of Data Messages Lost (Error Scenario, Random Topology)

As per the grid topology, Dynamic Source Routing is able to stabilize the network without losing a single data message. Directed Diffusion, on the other hand, does suffer the loss of a single data message before stabilization occurs. This outcome is expected for the same reasons explained above in the grid topology section.

5.7.4 Stabilization Time

The results for the last metric presented in this section, Stabilization Time, are displayed in Figure 43. In the closest error scenario, Directed Diffusion achieves the best Stabilization Time at a value of 0.47 s. Dynamic Source Routing obtains a higher value of 0.86 s. Comparably, in the intermediate error scenario, Directed Diffusion scores a Stabilization Time of 0.52 s while Dynamic Source Routing has a Stabilization Time of 0.63 s. Finally, for the farthest error scenario, Directed Diffusion requires 0.42 s for network stabilization while Dynamic Source Routing needs 0.66 s.

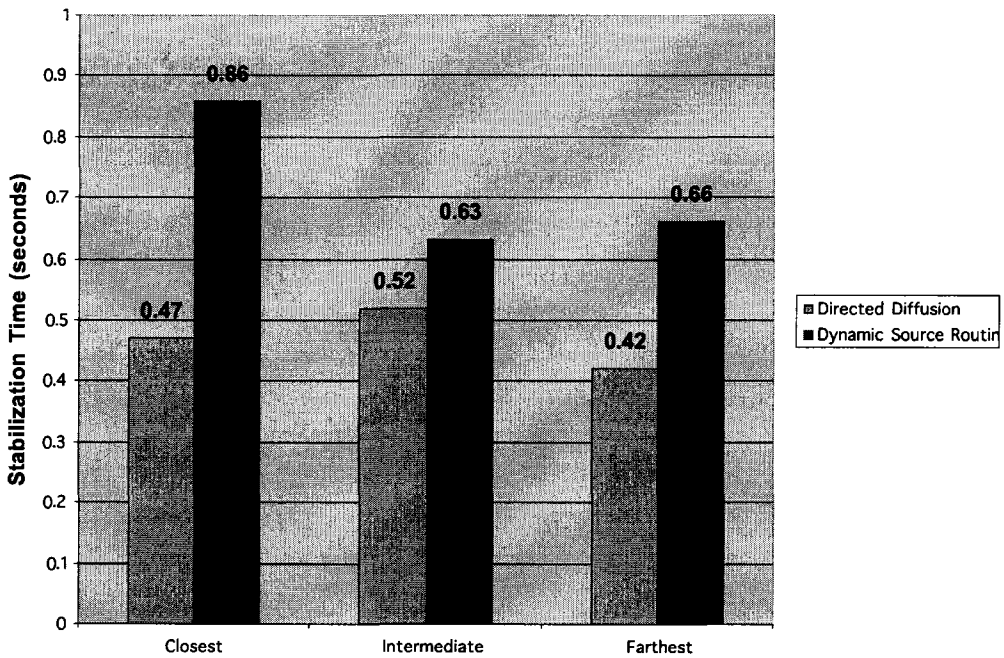


Figure 43 - Stabilization Time (Error Scenario, Random Topology)

As is evident in the above results, Directed Diffusion achieves the best Stabilization Time value for all experiments. This is due to the fact that Dynamic Source Routing has no additional cached routes to the base station in either of the three error scenarios. As

mentioned previously, this forces both protocols to flood the network in order to achieve stabilization, resulting in similar performances. Directed Diffusion has mildly better Stabilization Times due to the reasons detailed in the Stabilization Time results for the grid topology.

Chapter 6 – Conclusions and Future Work

6.1 Overview

In this chapter, conclusions are drawn as to which routing protocol should be selected for use in particular habitat monitoring situations. These conclusions are based on the results of the experimentations presented in the previous chapter. As well, an overview of the advantages of using the chosen metrics is presented. Finally, a summary of future work is given towards the end of the chapter that lays the groundwork for further research. Before recommendations are made, a summary of the advantages/disadvantages of each routing protocol is made available for reference in Table 6.

Protocol	Advantages	Disadvantages
Directed Diffusion	• Localized Interactions • Small memory footprint	• Interest flooding to correct errors • Duplicate data transmission
Dynamic Source Routing	• Routes discovered on demand • No periodic routing updates • Supports unidirectional links • Several paths can be cached to a destination	• Each packet must contain route record • Volatile networks require constant RRQ flooding
Minimum Transmission Routing	• Minimizes the expected number of retransmissions • No implicit error recovery cost	• Requires periodic routing update messages • Possible to lose data messages if the update interval is not small enough

Table 6- Routing Protocol Advantages/Disadvantages

6.2 Recommendations

As stated earlier, the primary goal of this work is to assist researchers in selecting the routing protocol best suited to the needs of their habitat monitoring application. It is important to note, however, that different sensor network applications may be more concerned with particular metrics. For example, a sensor network deployed for the purpose of object tracking may be more concerned with the Average Delay metric, ensuring that data events are received by the base station as quickly as possible. For another network deployed on a remote island, maximizing the network lifetime is critical, meaning that Average Dissipated Energy is a more pertinent metric. Before considering individual metrics, the routing protocol that performed best overall is discussed.

The protocol recommended for the majority of sensor network applications is Dynamic Source Routing. This is based on the performance of the protocol in the experiments detailed in Chapter 5.

In applications where network lifetime is critical, Dynamic Source Routing is the obvious choice. This is because it obtained the best values for the Average Dissipated Energy and Total Number of Messages Transmitted metrics by a wide margin in all of the Perfect Scenarios. As well, although it did not achieve the best score in the Average Delay experiments, it did attain values very close to those of the other routing protocols. In most habitat monitoring scenarios, an extra delay of 0.1 seconds is not considered significant, meaning that Dynamic Source Routing's performance is adequate. Similarly, although Dynamic Source Routing did not have the best value for Energy Consumption Variation, the differences between the three routing protocols for this metric are slim in all scenarios. As such, using Dynamic Source Routing does not significantly increase the risk of having certain sensor nodes fail more quickly due to excess energy consumption.

The choice of Dynamic Source Routing is further supported after examining the results of the Error Scenario experiments. Here, Dynamic Source Routing achieves the best Recovery Total Energy Consumption in both topologies amongst all types of failures, with the only exception being the farthest failure in the random topology. Even in this case, the results for both routing protocols are very close. Likewise, Dynamic Source Routing attains the lowest value for Total Number of Messages in all error scenarios.

Another important achievement of Dynamic Source Routing is that it did not lose a single data message during all network errors. This outcome is extremely relevant to error-prone sensor networks where losing even a single message is detrimental to the application (e.g., object tracking). It is possible, however, for Dynamic Source Routing to lose data messages during a network failure if it does not receive a Route Reply before its outgoing message buffer overflows.

Stabilization Time is the only error metric in which Dynamic Source Routing does not perform best under most error scenarios. Even though Directed Diffusion fares slightly better in this metric, the values attained by Dynamic Source Routing are still acceptable for the majority of habitat monitoring applications.

Another fact in support of Dynamic Source Routing is that its ability to cache multiple routes to the base station dramatically increases performance during network failures. In all error experiments where the data generating node is able to utilize another stored data path, Dynamic Source Routing outperforms Directed Diffusion in all metrics by a sizable margin.

Although Dynamic Source Routing performed admirably in the chosen scenarios, it is noteworthy to consider scenarios not included in the experimentation plan. In all of the experimentation scenarios, only a single node is generating data destined for the base station. This is a reasonable assumption for habitat monitoring applications because, most often, the user is only querying a small subset of nodes to receive data. Consider, however, the case where all nodes in the network (e.g., 36 nodes in a grid topology) are generating data to be routed to the base station; presumably because each of them is able to fulfill the specified interest. In this scenario, Dynamic Source Routing could perform more poorly because each node receiving the interest will need to generate a Route Request and flood it throughout the network in order to start routing data. Additionally, if a network error affects multiple nodes, each of these nodes will again need to re-broadcast a Route Request.

Directed Diffusion, however, should fare slightly better in this scenario. Initially, each node will send data back to the base station along (possibly) multiple data paths, resulting in a high volume of messages sent. Once the base station performs interest reinforcement, however, the number of data paths is reduced substantially. As well, if a network error occurs in this scenario, the base station re-broadcasts only a single interest message as usual. Further simulation studies would help in the analysis of this scenario.

Minimum Transmission Routing theoretically achieves the best performance when a large number of nodes are generating data. In contrast to Dynamic Source Routing, which must send more Route Request messages as the number of data generating nodes increases, Minimum Transmission Routing incurs no such overhead due to the use of a constantly updated tree-based topology. As well, if a network error

occurs, no additional messages need to be sent due to the periodic routing update messages. Therefore, Minimum Transmission Routing may be better equipped to handle habitat monitoring applications where a large number of nodes will be generating data for prolonged periods of time.

Additionally, even though Minimum Transmission Routing performed considerably worse in terms of energy consumption in all of the Perfect Scenario experiments, several cases can be made to advocate the use of this protocol in a habitat monitoring deployment. If Average Delay is the crucial metric in a sensor network application, Minimum Transmission Routing may be the proper choice of routing protocols. In the grid topology, for example, this protocol achieved the best Average Delay and placed a close second in the random topology experiment. This characteristic is useful in applications such as object tracking where receiving data events as soon as possible is paramount.

Another argument for using Minimum Transmission Routing is the error recovery mechanism built in to the protocol. Since there is no implicit cost associated with recovering from a network error in Minimum Transmission Routing, this protocol is well suited to sensor networks that are highly error-prone. This is important when sensor networks are deployed in an area where there are many potential sources of external interference (e.g., other electronic components). As mentioned earlier, however, it is still possible for Minimum Transmission Routing to lose data messages during a network failure if the routing update interval is larger than the data generation interval.

Based on the experimentation results, the only case where the use of Directed Diffusion can be advocated is in sensor network deployments where the amount of

memory available for the routing protocol is extremely limited. Directed Diffusion occupies approximately 1.5 KB less memory than Dynamic Source Routing and approximately 3 KB less than Minimum Transmission Routing. If the memory allocated to the routing protocol is a critical deployment issue, then the slight memory advantage of Directed Diffusion makes it the correct choice.

6.3 Choice of Metrics

In evaluating routing protocols for sensor networks, selecting pertinent metrics is one of the most significant decisions to make. Thus, the various metrics selected to compare the performance of the routing protocols is an important contribution of this thesis. Perhaps the most important metric for habitat monitoring is the Average Dissipated Energy – a custom metric that measures the amount of energy required to route a single data message back to the base station. Since energy dissipation is a prime concern in sensor networks, this metric is a good indicator when identifying which protocols can deliver data messages using the least amount of energy.

On a related note, the Total Number of Messages Transmitted metric is useful for determining which routing protocols minimize use of the radio. Since radio usage is the dominant source of energy dissipation, it is important to keep the number of messages sent to a minimum.

Energy Consumption Variation is also an important metric because it can be directly linked to network lifetime. If a subset of the nodes consumes energy more quickly than all other nodes in the network, this subset will fail sooner. Depending on the

location of the failed nodes, it is quite possible that some remaining nodes will no longer have a viable route to the base station.

Another important metric is Average Delay. Since some sensor network applications are time sensitive (e.g., object tracking), it is imperative that the base station receives data events as soon as possible. By measuring the Average Delay of each routing protocol, an indication of their performance in time sensitive applications is obtained.

The resource-constrained nature of sensor nodes also makes Code Size (both Program and Data) an important consideration. Even if a routing protocol implementation is found to perform well in all other metrics, the protocol cannot be used unless it fits reasonably into the memory available in the sensor nodes.

Evaluating how well routing protocols adapt to network failures is a necessity. As such, Recovery Total Energy Consumption is a relevant metric to determine how much energy is required to recover from a single network failure. If a sensor network is deployed under volatile conditions (i.e., the topology is very dynamic), minimizing the value for this metric is imperative to extending network lifetime.

Calculating the Number of Data Messages Lost is also a metric that deserves consideration. Losing data messages during a network failure is obviously an undesirable outcome. Consequently, a dynamic topology means that data may be frequently lost, negatively affecting sensor network applications.

Finally, Stabilization Time is a relevant metric when evaluating how well routing protocols adapt to network failures. During a network failure, it may be impossible for

data to be routed to the base station. Therefore, minimizing the amount of time required for the network to stabilize is essential for correct operation.

6.4 Future Directions

In concluding this thesis, it is useful to look at future directions that can be taken to expand on the research presented. The items listed in this section are items conceived during the course of the thesis that we did not have the opportunity to explore. This is either due to the fact that the item did not lie directly in the scope of the research or due to time constraints.

The first point of future research discussed is the addition of more scenarios to the experimentation plan. Currently, the experimentation plan includes scenarios that are relevant to the majority of habitat monitoring scenarios. As mentioned previously, this plan could be expanded to include scenarios where multiple sensor nodes, or perhaps all nodes, are sending data at the same time. Additionally, the error scenarios could be extended to encompass more choices. At present, these scenarios are concerned only with measuring the cost of a single network failure. Perhaps the impact of multiple concurrent failures could be measured to determine if this affects the resiliency of each routing protocol. It is worth noting that the novel framework for interest-based routing protocols presented in Chapter 4 already allows for multiple data senders in the sensor network due to the fact that all nodes receive the initial interest message. The sensing application code simply needs to be adjusted so that all nodes are able to generate data to fulfill the specified interest. Both the software framework and the routing protocol implementations are available from the author.

Another direction for future research is making modifications to the existing routing protocols in the hopes of improving performance. Currently, for example, Directed Diffusion has no implicit error detection mechanism, meaning that messages must be flooded each time an error occurs. Perhaps the concept of a Route Error message (similar to Dynamic Source Routing) can be applied to Directed Diffusion to minimize the cost of network errors.

Also, in the Dynamic Source Routing protocol, although a node can have multiple cached routes to the base station, there is no emphasis on ensuring that these routes are node or edge disjoint. Therefore, when a node or a link failure occurs, it may affect all cached routes, forcing the re-broadcasting of a Route Request message. By improving the Route Request algorithm to incorporate checks for node/edge disjoint paths, the protocol can minimize the chances of having to flood a Route Request when an error occurs. As shown in the experimentation results, Dynamic Source Routing performs extremely well when another cached route is utilized.

Another possible future consideration relates only to the Dynamic Source Routing protocol. In this protocol, the base station receives a Route Request containing the hops required to reach the base station from the source node. Assuming that links in the network are bi-directional, this implies that the base station can store the series of hops necessary to reach the source node and use them at a later time. This would allow the base station to directly communicate with data generating nodes without having to broadcast messages. For example, if the base station wishes to send an updated interest to a particular data generating node, it can use the stored hops to circumvent having to broadcast the interest message.

Finally, it is interesting to consider the possibility that the base station is within communications range of all nodes in the network (e.g., it can send data to any node in a single hop). This is not a common assumption in research involving routing protocols for sensor networks. As such, it is worthwhile to consider the impact that this assumption has on the routing protocols mentioned in this thesis. Also, it would be interesting to develop a novel routing protocol that uses this assumption to an operational advantage.

References

[1] D. Culler, , D. Estrin, et. al., "Overview of Sensor Networks," in *IEEE Computer Magazine*, Vol. 37, No. 8, 2004, pp. 41--49.

[2] I. Akyildiz, , W. Su, et. al., "A Survey on Sensor Networks," in *IEEE Communications Magazine*, Vol. 40, No. 8, 2002, pp. 102--114.

[3] A. Mainwaring, J. Polastre, et.al., "Wireless Sensor Networks for Habitat Monitoring," *Wireless Sensor Networks and Applications (WSNA '02)*, 2002.

[4] A. Tanenbaum, et. al., "Taking Sensor Networks From the Lab to the Jungle," in *IEEE Computer,* vol. 39, August 2006.

[5] A. Cerpa, "Habitat Monitoring: Application Driver for Wireless Communications Technology," *Proceedings of the ACM SIGCOMM Workshop on Data Communications*, 2001.

[6] E. Biagioni, B. Chee, K. Bridges,, University of Hawaii at Manoa, "A Remote Ecological Micro-Sensor Network," June 2000, http://www.botany.hawaii.edu/pods/overview.htm.

[7] Robotics and Intelligent Machines Laboratory, University of California at Berkeley, "Tracking vehicles with a UAV-delivered sensor network," 2001, http://robotics.eecs.berkeley.edu/~pister/29Palms0103/.

[8] C. Gamage et. al, "Security for the Mythical Air-Dropped Sensor Network," in *Proceedings of the 11th IEEE Symposium on Computers and Communications*, 2006.

[9] C. Sharp, S. Schaffert, et. al., "Design and implementation of a sensor network system for vehicle tracking and autonomous interception," *Proceedings of the Second European Workshop on Wireless Sensor Networks*, 2005.

[10] N. Xu, S. Rangwala, et.al, "A Wireless Sensor Network for Structural Monitoring," *Proceedings of the ACM Conference on Embedded Networked Sensor Systems,* 2004.

[11] C. Perkins, "Ad-hoc On-Demand Distance Vector Routing", *MILCOM '97 Panel on Ad Hoc Networks*, 1997.

[12] C. Perkins, P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers," in *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, 1994, pages 234--244.

[13] C. Karlof, Y. Li, J. Polastre, "ARRIVE: Algorithm for Robust Routing in Volatile Environments," University of California at Berkeley, Berkeley, CA, United States, Tech. Rep. UCB/CSD-03-1233, 2003.

[14] K. Akkaya, et. al, "A Survey on Routing Protocols for Wireless Sensor Networks," in *Journal of Ad Hoc Networks*, vol. 3, May 2005.

[15] J. Hill, "System Architecture for Wireless Sensor Networks," Ph.D. dissertation, University of California, Berkeley, CA, United States, 2003.

[16] P. Levis, N. Lee, et.al., "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[17] B. Titzer, "AVRORA: The AVR Simulation and Analysis Framework," M.S. thesis, University of California, Los Angeles, CA, United States, 2004.

[18] C. Intanagonwiwat, "Directed Diffusion: An Application-Specific and Data-Centric Communication Paradigm for Wireless Sensor Networks," Ph.D. dissertation, University of Southern California, Los Angeles, CA, United States, 2002.

[19] Y. Yu, R. Govindan, D. Estrin, " Geographical and Energy Aware Routing: a recursive data dissemination protocol for wireless sensor networks," UCLA Computer Science Department, Los Angeles, CA, United States, Tech. Rep. UCLA/CSD-TR-01-0023, 2001.

[20] K. Romer, "The Lighthouse Location System for Smart Dust," in *Proceedings of MobiSys*, May 2003.

[21] V. Ramaduria, M. Sichitiu, "Localization in Wireless Sensor Networks: A Probabilistic Approach," in *Proceedings of the 2003 International Conference on Wireless Networks*, June 2003.

[22] S. Brennan, A. Maccabe, et. al, "Radiation Detection with Distributed Sensor Networks," *IEEE Computer,* vol. 37, August 2004.

[23] D. Ganesan, R. Govindan, et. al., "Highly-Resilient, Energy-Efficient Multipath Routing in Wireless Sensor Networks," *Mobile Computing and Communications Review,* vol. 4, no. 5, October 2001.

[24] D. Johnson, D. Maltz, J. Broch, " DSR: The Dynamic Source Routing Protocol for Multi-Hop Wireless Ad Hoc Networks," *Ad Hoc Networking,* C. Perkins, Ed. Addison Wesley, 2001, pp. 139--172.

[25] A. Woo, "A Holistic Approach to Multihop Routing in Sensor Networks," Ph.D. dissertation, University of California, Berkeley, CA, United States, 2004.

[26] E. Demaine, A. Lopez-Ortiz, J. I. Munro, "Frequency estimation of internet packet streams with limited space," in *Proceedings of the European Symposium on Algorithms,* 2002, pp. 348--360.

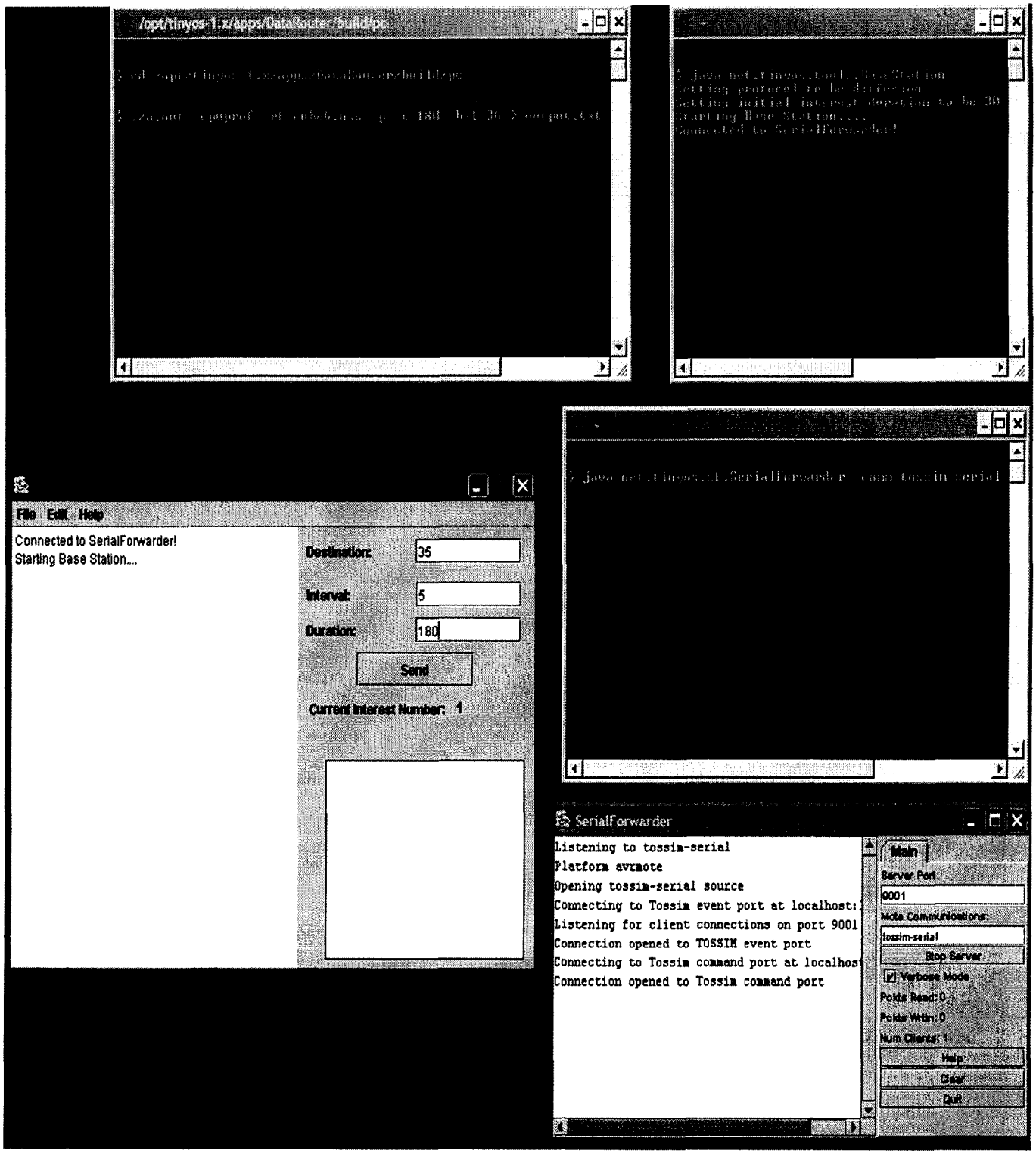
[27] V. Shnayder, M. Hempstead, et. al., " Simulating the Power Consumption of Large-Scale Sensor Network Applications," in *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems,* 2004, pp. 188--200.

[28] M. Demmer, P. Levis, et. al., "Tython: A Dynamic Simulation Environment For Sensor Networks," 2004

[29] J. Elson, D. Estrin, "Time synchronization for wireless sensor networks," *IPDPS Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing*, 2001.

[30] D. Gay, P. Levis, et. al., "The nesC Language: A Holistic Approach to Networked Embedded Systems," in *Proceedings of Programming Language Design and Implementation*, 2003.

Appendix A: Experimentation Setup



Appendix B: TOSSIM Grid Topology File

```
0:1:0.00
0:6:0.00
1:2:0.00
1:0:0.00
1:7:0.00
2:3:0.00
2:1:0.00
2:8:0.00
3:4:0.00
3:2:0.00
3:9:0.00
4:5:0.00
4:3:0.00
4:10:0.00
5:4:0.00
5:11:0.00
6:7:0.00
6:12:0.00
6:0:0.00
7:8:0.00
7:6:0.00
7:13:0.00
7:1:0.00
8:9:0.00
8:7:0.00
8:14:0.00
8:2:0.00
9:10:0.00
9:8:0.00
9:15:0.00
9:3:0.00
10:11:0.00
10:9:0.00
10:16:0.00
10:4:0.00
11:10:0.00
11:17:0.00
11:5:0.00
12:13:0.00
12:18:0.00
12:6:0.00
13:14:0.00
13:12:0.00
13:19:0.00
13:7:0.00
14:15:0.00
14:13:0.00
14:20:0.00
14:8:0.00
15:16:0.00
15:14:0.00
15:21:0.00
15:9:0.00
16:17:0.00
16:15:0.00
16:22:0.00
16:10:0.00
17:16:0.00
17:23:0.00
17:11:0.00
18:19:0.00
```

```
18:24:0.00
18:12:0.00
19:20:0.00
19:18:0.00
19:25:0.00
19:13:0.00
20:21:0.00
20:19:0.00
20:26:0.00
20:14:0.00
21:22:0.00
21:20:0.00
21:27:0.00
21:15:0.00
22:23:0.00
22:21:0.00
22:28:0.00
22:16:0.00
23:22:0.00
23:29:0.00
23:17:0.00
24:25:0.00
24:30:0.00
24:18:0.00
25:26:0.00
25:24:0.00
25:31:0.00
25:19:0.00
26:27:0.00
26:25:0.00
26:32:0.00
26:20:0.00
27:28:0.00
27:26:0.00
27:33:0.00
27:21:0.00
28:29:0.00
28:27:0.00
28:34:0.00
28:22:0.00
29:28:0.00
29:35:0.00
29:23:0.00
30:31:0.00
30:24:0.00
31:32:0.00
31:30:0.00
31:25:0.00
32:33:0.00
32:31:0.00
32:26:0.00
33:34:0.00
33:32:0.00
33:27:0.00
34:35:0.00
34:33:0.00
34:28:0.00
35:34:0.00
35:29:0.00
```

Appendix C: TOSSIM Log File Example

```
35: APP: Data ready to be sent: (372) (0:0:10.33077750)
35: Sending RRQ to request route to node 0
35: Successfully cached RRQ message
35: xxxSending rrq message to: 65535
35: MAC: Setting Send Timer for 15 milliseconds
35: Successfully queued RRQ message for transmission at time 0 seconds
35: APP: Successfully sent sensor data message
35: MAC: Sending RRQ message to node 65535 at time 0 milliseconds
35: MAC: Successfully sent RRQ message
35: POWER: Mote 35 RADIO_STATE TX at 41414842
34: POWER: Mote 34 RADIO_STATE RX at 41423823
29: POWER: Mote 29 RADIO_STATE RX at 41423861
29: POWER: Mote 29 RADIO_STATE TX at 41492461
34: POWER: Mote 34 RADIO_STATE TX at 41492461
35: POWER: Mote 35 RADIO_STATE RX at 41493242
29: POWER: Mote 29 RADIO_STATE TX at 41493261
34: POWER: Mote 34 RADIO_STATE TX at 41493261
29: POWER: Mote 29 RADIO_STATE TX at 41494061
34: POWER: Mote 34 RADIO_STATE TX at 41494061
35: POWER: Mote 35 RADIO_STATE RX at 41494092
29: POWER: Mote 29 RADIO_STATE TX at 41494861
34: POWER: Mote 34 RADIO_STATE TX at 41494861
35: POWER: Mote 35 RADIO_STATE RX at 41494892
29: POWER: Mote 29 RADIO_STATE TX at 41495661
34: POWER: Mote 34 RADIO_STATE TX at 41495661
35: POWER: Mote 35 RADIO_STATE RX at 41495692
29: POWER: Mote 29 RADIO_STATE RX at 41496461
29: Received new RRQ message at time: 0
29: Processing RRQ message with id 1 from node 35
29: Successfully cached RRQ message
29: xxxSending rrq message to: 65535
29: MAC: Setting Send Timer for 105 milliseconds
29: Successfully queued RRQ message for transmission at time 0 seconds
34: POWER: Mote 34 RADIO_STATE RX at 41496461
34: Received new RRQ message at time: 0
34: Processing RRQ message with id 1 from node 35
34: Successfully cached RRQ message
34: xxxSending rrq message to: 65535
34: MAC: Setting Send Timer for 30 milliseconds
34: Successfully queued RRQ message for transmission at time 0 seconds
35: POWER: Mote 35 RADIO_STATE RX at 41496412
35: POWER: Mote 35 RADIO_STATE RX at 41496492
35: Successfully broadcasted RRQ message!
22: POWER: Mote 0 CPU_CYCLES 17343.5 at 41600597
22: POWER: Mote 1 CPU_CYCLES 17139.5 at 41600597
22: POWER: Mote 2 CPU_CYCLES 16466.0 at 41600597
22: POWER: Mote 3 CPU_CYCLES 16371.5 at 41600597
22: POWER: Mote 4 CPU_CYCLES 15559.0 at 41600597
22: POWER: Mote 5 CPU_CYCLES 12768.0 at 41600597
22: POWER: Mote 6 CPU_CYCLES 17139.5 at 41600597
22: POWER: Mote 7 CPU_CYCLES 19489.0 at 41600597
22: POWER: Mote 8 CPU_CYCLES 19533.5 at 41600597
22: POWER: Mote 9 CPU_CYCLES 19439.0 at 41600597
```

Appendix D: Java Program to Calculate Average Delay

```
package net.tinyos.tools;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.StringTokenizer;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

/**
 * Class that calculates values for delay-related metrics using
 * a TOSSIM log file
 *
 * @author Brett
 *
 */

public class DelayLogTool {

	private static Pattern dataSentPattern;

	private static Matcher dataSentMatcher;

	private static Pattern dataReceivedPattern;

	private static Matcher dataReceivedMatcher;

	private static String DATA_SENT_REGEX = "APP: Data";

	private static String DATA_RECEIVED_REGEX = "Base Station received new data";

	public static void main(String args[]) throws IOException {

		// Ask the user for the location of the TOSSIM log file

		String str = null;

		File logFile = null;

		System.out.print("Enter the location of the TOSSIM log file: ");

		str = getInput();

		System.out.println();

		if (str == null) {
			System.out.println("Log file path was null!");
			return;
		}

		// Validate the existence of this file
		logFile = new File(str);

		if (!logFile.exists()) {
			System.out.println("Invalid log file path!");
			return;
		}

		//C:/eclipse/workspace/LogTools/bin/mt_delay.txt
```

```
		calculateAverageDelay(logFile);
}

/**
 * Program works as follows:
 * 1) Find all occurances of the application sending a message.
 * 2) Parse out the data being sent and the time it is being sent at, and store it in a hashmap
 * 3) Parse out the data being received and place it in the hashmap
 *  - First value in the hashmap is the time at which the message was sent.
 *
 * All other values are the times at which the message was received by the
 * base station. Hashmap looks like:
 *
 * Data | Times (ArrayList) ---- ----- 546 [SendingTime][Rx Time 1][Rx Time 2][.....]
 *
 * @param file
 * @throws FileNotFoundException
 * @throws IOException
 */
private static void calculateAverageDelay(File file){

		FileReader fopen;
		try {
			fopen = new FileReader(file);
		} catch (FileNotFoundException e) {
			System.out.println("Unable to open log file for reading: " + e);
			return;
		}

		int numMessagesSent = 0;

		//Initialize the REGEX patterns

		dataSentPattern = Pattern.compile(DATA_SENT_REGEX);
		dataReceivedPattern = Pattern.compile(DATA_RECEIVED_REGEX);

		String line = null;

		HashMap delayMap = new HashMap();

		// Read in all lines and create a hashmap
		BufferedReader br = null;
		try {
			br = new BufferedReader(fopen);
			while ((line = br.readLine()) != null) {
				dataSentMatcher = dataSentPattern.matcher(line);
				dataReceivedMatcher = dataReceivedPattern.matcher(line);
				if (dataSentMatcher.find()) {
					// Parse the data being sent
					StringTokenizer tok = new StringTokenizer(line, "()");
					tok.nextToken();

					String data = tok.nextToken();

					// Parse the time it is being sent
					tok.nextToken();
					String time = tok.nextToken();

					// Insert into the hash map
					if (delayMap.get(data) == null) {
						ArrayList list = new ArrayList();
						list.add(time);
						delayMap.put(data, list);
					} else {
						System.out.println("Data " + data
								+ " already existed in the hash map");
						return;
					}
```

```
                } else if (dataReceivedMatcher.find()) {
                        // Parse the data being received
                        StringTokenizer tok = new StringTokenizer(line, "()");
                        tok.nextToken();

                        String data = tok.nextToken();

                        // Parse the time it is being sent
                        tok.nextToken();

                        String time = tok.nextToken();

                        // Place the time in the hashmap
                        if (delayMap.get(data) == null) {
                                System.out.println("Could not find data " + data
                                                + " in the hash map");
                                return;
                        } else {
                                ArrayList list = (ArrayList) delayMap.get(data);
                                list.add(time);
                                delayMap.put(data, list);
                        }

                } else {
                        System.out
                                        .println("Line did not match any regular expressions: "
                                                        + line);
                        return;
                }
        }
} catch (IOException e) {
        System.out.println("Encountered IO error while reading log file: " + e);
        return;
}
finally{
        try {
                br.close();
                fopen.close();
        } catch (IOException e) {
                System.out.println("Unable to close input streams: " + e);
        }


}

// printDelayMap(delayMap);

double totalAverageDelay = 0;

// Calculate average delay
Iterator itr = delayMap.keySet().iterator();
while (itr.hasNext()) {
        double averageDelay = 0;
        String key = (String) itr.next();
        ArrayList list = (ArrayList) delayMap.get(key);
        String sendTime = (String) list.get(0);

        for (int i = 1; i < list.size(); i++) {
                // Calculate delay for this entry
                double delay = 0;
                String receiveTime = (String) list.get(i);
                delay = calculateDelay(sendTime, receiveTime);

                averageDelay += delay;
        }
        if (list.size() > 1) {
                averageDelay = averageDelay / (double) (list.size() - 1);

                System.out.println(key + ": " + averageDelay);
```

```
                        numMessagesSent++;
                } else {
                        System.out.println("Data " + key
                                        + " was not received by the base station");
                }

                totalAverageDelay += averageDelay;
        }

        totalAverageDelay = totalAverageDelay
                        / (double) (delayMap.keySet().size());
        System.out.println("Total Number of messages sent: " + numMessagesSent);
        System.out.println("Total Average Delay: " + totalAverageDelay);


}

/**
 * Debugging method that allows the data structure storing the delay
 * information to be represented visually
 *
 * @param delayMap
 */
private static void printDelayMap(HashMap delayMap) {
        // Print the hashmap
        Iterator itr = delayMap.keySet().iterator();
        while (itr.hasNext()) {
                String key = (String) itr.next();
                ArrayList list = (ArrayList) delayMap.get(key);
                System.out.println(key + ": " + list.toString());
        }
}

/**
 * 0:0:25.88187050
 *
 * @param startTime
 * @param endTime
 * @return
 */
private static double calculateDelay(String startTime, String endTime) {
        double diff = 0;
        double startMinutes = 0;
        double startSeconds = 0;
        double endMinutes = 0;
        double endSeconds = 0;

        StringTokenizer tok = new StringTokenizer(startTime, ":");
        tok.nextToken();

        startMinutes = Double.parseDouble(tok.nextToken().trim());
        startSeconds = Double.parseDouble(tok.nextToken().trim());

        tok = new StringTokenizer(endTime, ":");
        tok.nextToken();

        endMinutes = Double.parseDouble(tok.nextToken().trim());
        endSeconds = Double.parseDouble(tok.nextToken().trim());

        diff = (endMinutes * 60 + endSeconds)
                        - (startMinutes * 60 + startSeconds);

        return diff;
}

/**
 * Get input from the command line
 *
 * @param str
 * @return
```

```
        */
        private static String getInput() {
                String str = null;

                try {
                        BufferedReader in = new BufferedReader(new InputStreamReader(
                                        System.in));
                        str = in.readLine();
                } catch (IOException e) {
                        System.out.println("Unable to read input from the command line: "
                                        + e);
                }
                return str;
        }
}
```

Appendix E: Java Program for Energy Consumption Metrics

```
package net.tinyos.tools;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;
import de.pxlab.stat.Stats;

/**
 * Compute various energy consumption-related metrics given a PowerTOSSIM log
 * file
 *
 * @author Brett
 *
 */

public class PowerLogTool {

	public static void main(String args[]) throws IOException {

		// Ask the user for the location of the PowerTOSSIM log file

		String str = null;
		int numberOfMessages = 0;

		File logFile = null;

		System.out.print("Enter the location of the PowerTOSSIM log file: ");

		str = getInput();

		System.out.println();

		if (str == null) {
			System.out.println("Log file path was null!");
			return;
		}

		// Validate the existence of this file
		logFile = new File(str);

		if (!logFile.exists()) {
			System.out.println("Invalid log file path!");
			return;
		}

		// Ask the user for the number of data messages received during the
		// simulation

		System.out.print("Enter the number of unique data messages received: ");

		str = getInput();

		System.out.println();

		if (str == null) {
			System.out.println("Number of messages was null!");
			return;
		}

		// Parse the string to an integer
```

```
		try {
			numberOfMessages = Integer.parseInt(str);

			if (numberOfMessages <= 0)
				throw new Exception();

		} catch (Exception e) {
			System.out
					.println("Could not parse number of messages from string: "
							+ str);
			return;
		}

		// Now compute the energy consumption values

		computeAverageDisspatedEnergy(logFile, numberOfMessages);
}

/**
 * Get input from the command line
 *
 * @param str
 * @return
 */
private static String getInput() {
		String str = null;

		try {
			BufferedReader in = new BufferedReader(new InputStreamReader(
					System.in));
			str = in.readLine();
		} catch (IOException e) {
			System.out.println("Unable to read input from the command line: "
					+ e);
		}
		return str;
}

/**
 * Compute the average dissipated energy given a PowerTOSSIM log file and
 * the number of data messages received during the simulation
 *
 * @param file
 * @throws FileNotFoundException
 * @throws IOException
 */
private static void computeAverageDisspatedEnergy(File file,
			int numberOfMessagesReceived){
		int numNodes = 0;
		double totalCpuPower = 0;
		double totalRadioPower = 0;
		double powerArr[] = new double[36];

		FileReader fopen;
		try {
			fopen = new FileReader(file);
		} catch (FileNotFoundException e1) {
			System.out.println("Unable to locate log file: " + e1);
			return;
		}

		BufferedReader br = null;
		try {
			br = new BufferedReader(fopen);
			String line = null;
			while ((line = br.readLine()) != null) {

				// Is this a cpu cycle line?
				if (line.indexOf("cpu_cycle total") >= 0) {
					// Get the node ID
```

```
                                StringTokenizer nodeTok = new StringTokenizer(line, ", ");
                                nodeTok.nextToken();
                                int nodeID = Integer.parseInt(nodeTok.nextToken());

                                // Get the total cpu power consumed for this node
                                StringTokenizer powerTok = new StringTokenizer(line, ":");
                                powerTok.nextToken();
                                String powerTotal = powerTok.nextToken();

                                double power = Double.parseDouble(powerTotal.trim());

                                totalCpuPower += power;

                                // Add this power to the array position for this node
                                powerArr[nodeID] += power;

                                numNodes++;
                        } else if (line.indexOf("radio total") >= 0) { // Is this a

                        // radio
                                // total line
                                // Get the node ID
                                StringTokenizer nodeTok = new StringTokenizer(line, ", ");
                                nodeTok.nextToken();
                                int nodeID = Integer.parseInt(nodeTok.nextToken());

                                // Get the total radio power consumed for this node
                                StringTokenizer powerTok = new StringTokenizer(line, ":");
                                powerTok.nextToken();
                                String powerTotal = powerTok.nextToken();

                                double power = Double.parseDouble(powerTotal.trim());

                                totalRadioPower += power;

                                // Add this power to the array position for this node
                                powerArr[nodeID] += power;

                        }

                }
        } catch (NumberFormatException e) {
                System.out
                                .println("Number Format Error parsing PowerTOSSIM log file: "
                                                + e);
                return;
        } catch (IOException e) {
                System.out
                                .println("IO Error while accessing PowerTOSSIM log file: "
                                                + e);
                return;
        } finally {
                try {
                        br.close();
                        fopen.close();
                } catch (IOException e) {
                        System.out.println("Unable to close input streams: " + e);
                }

        }

        System.out.println("Total Number of Nodes: " + numNodes);
        System.out.println("Total Number of Messages Received: "
                        + numberOfMessagesReceived);
        System.out.println("Total CPU Power Dissipated =  " + totalCpuPower);
        System.out.println("Average CPU Power Dissipated = "
                        + (totalCpuPower / numberOfMessagesReceived));
        System.out.println("Total Radio Power Dissipated = " + totalRadioPower);
        System.out.println("Average Radio Power Dissipated = "
                        + (totalRadioPower / numberOfMessagesReceived));
```

```
                System.out.println("Total Power Consumed Dissipated = "
                                + (totalRadioPower + totalCpuPower));
                System.out
                                .println("Average Total Power Dissipated = "
                                                + ((totalRadioPower + totalCpuPower) /
numberOfMessagesReceived));

                double variance = Stats.variance(powerArr);
                double standardDeviation = Stats.standardDev(powerArr);

                System.out.println("Variance: " + variance);
                System.out.println("Standard Deviation: " + standardDeviation);

        }
}
```