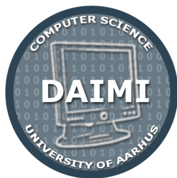
Evaluating Wireless Sensor Network Power Consumption:

A Case Study on Two Routing Protocols for Structural Health Monitoring

By:
Elvar Þór Ólafsson (20030582)
Henrik Skaarup Andersen (20030577)

Supervisor:
Lars Michael Kristensen



Master's Thesis
Department of Computer Science - Daimi
University of Aarhus
August 14, 2008

Contents

1 Introduction **1**
- 1.1 Statement . . . 3
- 1.2 Readers guide . . . 3

2 Background **5**
- 2.1 Structural Health Monitoring . . . 5
  - 2.1.1 Sensor Networks in SHM . . . 6
- 2.2 SensoByg Project . . . 7
  - 2.2.1 Specification and Requirements . . . 8
  - 2.2.2 Our Choice of Routing Protocols . . . 8
- 2.3 Routing Protocols . . . 9
  - 2.3.1 Data-centric Protocols . . . 9
  - 2.3.2 Hierarchial Protocols . . . 10
  - 2.3.3 Location-based Protocols . . . 13

3 Description of Protocols **15**
- 3.1 Geographical Adaptive Fidelity . . . 15
  - 3.1.1 Grid Overlay . . . 16
  - 3.1.2 Node States . . . 17
  - 3.1.3 Parameters and Node Ranking . . . 18
  - 3.1.4 Routing in GAF . . . 19
- 3.2 Autonomous Multicast-tree Creation Algorithm . . . 20
  - 3.2.1 Initial Tree Creation . . . 21
  - 3.2.2 Tree Maintenance . . . 23

4 Implementation Platform **26**
- 4.1 TinyOS 2 . . . 26
  - 4.1.1 Interfaces and Components . . . 27
  - 4.1.2 Configurations . . . 29
- 4.2 TOSSIM . . . 29
  - 4.2.1 Running an Application in TOSSIM . . . 30
  - 4.2.2 Using Debug Statements in TOSSIM . . . 32
  - 4.2.3 Our Experience with TOSSIM . . . 33
- 4.3 Power Simulation . . . 34
  - 4.3.1 Energy Calculations . . . 34
- 4.4 System Architecture . . . 36
  - 4.4.1 Overview . . . 37

4.4.2 Router Component 37
4.4.3 MessageSender Component 40
4.4.4 RouteSelector Component 40
4.4.5 Communication Component 42
4.4.6 EnergyModel Component 42
4.5 Implementation of the Routing Protocols 43
4.5.1 Time Synchronization 44
4.5.2 Base Station 44
4.5.3 GAF 44
4.5.4 AMCA 46

5 Experimental Setup **49**
5.1 Setup 49
5.2 Performance Metrics 51
5.2.1 GAF Specific Experiments and Adaptations 53
5.2.2 AMCA Specific Experiments and Adaptations 54

6 Experimental Results **58**
6.1 GAF 58
6.1.1 Scenario 1 59
6.1.2 Scenario 2 62
6.1.3 Scenario 3 64
6.1.4 Scenario 4 66
6.1.5 Scenario 5 68
6.1.6 Conclusions on GAF 71
6.2 AMCA 72
6.2.1 Scenario 1 73
6.2.2 Scenario 2 74
6.2.3 Scenario 3 76
6.2.4 Scenario 4 79
6.2.5 Scenario 5 81
6.2.6 Conclusion on AMCA 83
6.3 Comparison of the Protocols 84

7 Conclusion and Future Work **89**
7.1 Future Work 90
7.1.1 GAF 91
7.1.2 AMCA 91

Bibliography **93**

Abstract

Structural health monitoring (SHM) has recently become a research topic which has gathered a lot of attention. SHM is concerned with monitoring the structural state, such as temperature and moisture, in structures ranging from buildings to bridges. Until now, the technology used to monitor the state of structures has been based on sensors embedded inside the structure which had to be wired together. This means that setting up the network of sensors is a time-consuming task. Coupled with the cost of the sensors, SHM has become a costly affair, and hence it has not seen much practical use.

In recent years though, wireless network technology has matured greatly. It has now become possible to construct small wireless sensors, which can serve the same purpose as their wired counterparts. These wireless sensor units are easier to deploy, which makes them a more cost-effective solution to SHM.

Switching to wireless technology introduces a number of issues which must be considered. The most immediate concern arises from the fact that wireless sensor units embedded in a structure, do not have access to any external power supply. This makes them dependent on batteries, and hence the power consumption of the units must be carefully considered, such that the lifetime of the SHM sensor network can be extended for as long as possible. At the same time it is important to be able to scale the sensor network so that it is possible to perform SHM on very large structures. This raises the issue of radio ranges of the wireless sensor units, which may not be very large. Therefore, it becomes necessary to consider routing protocols for the sensor units, which will forward the sensed information to its destination while keeping power consumption as low as possible.

In this thesis we investigate these issues in the context of the SensoByg project which is a research project focussing on SHM. We implement two routing protocols: The Autonomous Multicast-tree Creation Algorithm (AMCA) and the Geographic Adaptive Fidelity (GAF). Using simulation, we evaluate the implementations in the context of the SensoByg project, with special focus on the ability of each protocol to extend the lifetime of the wireless sensor network. Finally, we conclude upon each protocols general applicability in SHM.

1 Introduction

Society today continues to build larger and more complex infrastructures as engineering techniques and the quality of materials improve. But even the most advanced buildings still decay over time, as they are subject to exterior forces, such as seismic events and weather phenomena. It is therefore necessary to be able to inspect and evaluate the structural state of these structures, to be able to determine weak spots or damaged areas. As the size and complexity of the structures increases, this becomes a difficult and time consuming task.

Structural Health Monitoring (SHM), is a fairly new technology concerned with collecting and evaluating information pertaining to the state of structures. The idea is that the cumbersome manual task of inspecting a large structure, can be done automatically by a number of sensor units (often referred to as *nodes*) placed strategically in various places on the structure. These nodes contain sensors that can read values such as humidity and temperature. The values are then transmitted from each node to a central location (often called the *base station*), where they can be used to evaluate the state of the structure being monitored. By making the process of inspecting the structure automatic, it is possible to follow the structural state much closer, as readings from the sensors can arrive on a more frequent basis. This may reduce the cost of repairing any damages, as it is possible to discover weaknesses earlier.

Until now, SHM systems have not seen much practical use. Mainly because early systems have been based on arrays of sensors which had to be wired together. This is an expensive and time consuming task, and the costs therefore remain greater than the reward. However, recent advances in sensor, radio and microprocessor technologies have opened up a new approach to SHM. It is now becoming possible to manufacture small, reliable sensor units which use wireless communication. These nodes can be placed on, or embedded in, a structure, to create a *Wireless Sensor Network* (WSN). This makes the

task of deploying and maintaining the nodes a much less expensive activity.

The switch to wireless communication introduces a number of problems which must be addressed, in order for SHM to truly become practical. Namely, radio ranges, durability, and energy consumption. As it can be desirable to embed wireless sensors inside the materials that a structure is made up of, it becomes important that the sensor units can sustain themselves in such harsh environments. Also, as there is often no way to access such nodes, the radios in use must be strong enough to be able to communicate with the outside world. With the recent advances that manufacturers[1] of such nodes are making, this is becoming increasingly possible.

The problem of the energy consumption of the nodes, is also one that has recently received a lot of attention from the research community. The problem itself arises from the fact that a wireless sensor unit, will not have access to any external power supplies. It will therefore have to rely on energy sources such as batteries. As it is not always possible to access the nodes after deployment, they will at some point run out of energy, marking the end of the WSNs lifetime. To extend the lifetime of the network long enough, so that it becomes useful in a SHM setting, it is necessary to consider the energy consumption of the nodes carefully.

Currently, a danish research project called *SensoByg*[4], is investigating the applicability of wireless sensors in SHM. Early results from this project show, that while it is possible to construct nodes that can cope with the harsh environments they can be subjected to, expected radio ranges for wireless communication exceeding 5-10 meters may still be optimistic. This introduces the problem of collecting the sensor readings from each node and getting them to the base station.

In the light of this, it becomes necessary to consider routing protocols that are able to route messages containing sensor readings, from a node to the base station, by sending it through intermediate nodes, one *hop* at a time. Routing protocols already exists that are able to achieve this for many different settings. However, as described above, the nodes in a WSN are severely energy constrained. Therefore it is a necessity that such a routing protocol is designed to consume as little energy as possible, while still being reliable. The challenge arises from the fact that a protocol should be able to balance the load of the network such that no single node will deplete its energy much earlier than the rest of the nodes in the network. In other words, it should be able to utilize all the available energy in the network.

In this thesis we implement and compare two routing protocols, which try to address the specific challenges found in WSNs using two different approaches: The *Geographic Adaptive Fidelity[16] (GAF)* protocol, and the *Autonomous Multicast-tree Creation Algorithm[15] (AMCA)* protocol. GAF is a location aware routing scheme, which creates an overlay network in order to divide the WSN into manageable regions. AMCA is a cluster and tree based protocol. Both of these protocols take advantage of advanced sleep states in order to prolong the lifetime of the network.

The evaluation of the protocols are done in context of the early requirements and one of the scenarios described in the SensoByg project. The scenario concerns monitoring the structural health of a bridge using a WSN. The goal is to evaluate the general applicability of each of the protocols in a real-world SHM setting, and to possibly identify any weaknesses in the WSN approach.

1.1 Statement

In this thesis we implement and evaluate two routing protocols: The *Geographic Adaptive Fidelity (GAF)*[16] protocol and the *Autonomous Multicast Tree Creation (AMCA)*[15] protocol. The implementation will be done in TinyOS 2[5], which is an open-source source operating system designed specifically for WSNs.

The protocols will be evaluated for their performance in a SHM scenario. The scenario is based on a case study from the danish research project SensoByg[4], which focuses on SHM. This project and the cases it is concerned with, will also be investigated and described.

The goal is to determine each protocols ability to extend the lifetime of a WSN in the given scenario, in order to determine the protocols applicability within the area of SHM. This is done by simulating each protocol in TOSSIM[12], which is a TinyOS simulator. The simulations will be based on the SHM scenarios, and will be evaluated with respect to early requirements put forth in the SensoByg project. As the project aims for lifetimes of five years or more, in settings requiring very large scale WSNs, the primary measure for performance will be the ability of each protocol to extend the WSNs lifetime. The implementations of the protocols will also be evaluated on their ability to route messages in the network with minimal loss and overhead. To evaluate each of the protocols, we have created six different scenarios that are based on the scenario from the SensoByg project.

1.2 Readers guide

We now describe the contents of this thesis along with who of the two authors, Elvar Þór Ólafsson (EÞÓ) and Henrik Skaarup Andersen (HSA), is the main responsible for each of the sections. In general, HSA is responsible for sections concerning GAF, while EÞÓ is responsible for sections concerning AMCA. If a section has not been assigned a responsible author, the responsibility for that section is shared.

Chapter 2: Background covers the background information needed to understand this thesis. First we describe Structural Health Monitoring and how sensor networks are used for that purpose. We then move onto describing the Sensobyg project and how we use that as a background for our thesis. Finally, we describe other routing protocols used in wireless sensor networks today and why we have not chosen them as our implementation protocols. HSA is responsible for section 2.1 while EÞÓ is responsible for sections 2.2 and 2.3.

Chapter 3: Protocol Description covers the description of the GAF and AMCA routing protocols, which we investigate in this thesis. HSA is responsible for section 3.1 while EÞÓ is responsible for section 3.2.

Chapter 4: Implementation Platform covers the implementation of the routing protocols. First we describe the TinyOS programming platform that we used. Then we

describe the TOSSIM simulator that we used for our simulation purposes and how we implemented our own power modeling for use in the simulations. This includes our experience in dealing with TOSSIM. Finally, we have a description of the framework that we used as a common basis for the routing protocols along with any specific implementation details for both routing protocols. HSA is responsible for sections 4.1, 4.4, and 4.5.3. EÞÓ is responsible for sections 4.2, 4.3, and 4.5.4.

Chapter 5: Experimental Setup describes how we came up with the performance metrics that we are measuring in the simulations and how the simulation setup was done. We then go on to describe routing protocol specific details that we measure during the simulations.

Chapter 6: Results is where we list the results from each of our scenarios that we ran in our simulations and analyze them with regards to the performance metrics specified in chapter 5. We then describe how the two protocols compare to each other. HSA is responsible for section 6.1 while EÞÓ is responsible for section 6.2.

Chapter 7: Conclusion and Future Work covers the conclusions that we have reached after having analyzed the results from our simulations. We then discuss issues that could be worked on in the future. HSA is responsible for section 7.1.1 while EÞÓ is responsible for section 7.1.2.

Implementation: HSA is responsible for the framework and the GAF implementation and EÞÓ is responsible for the AMCA implementation. All the code related to this thesis can be found online at `http://www.daimi.au.dk/~hsa/thesis/` Also, a *readme* file can be found, which explains how to setup and run our simulations.

2
Background

In section 2.1, we introduce Structural Health Monitoring (SHM) and some of the challenges to be addressed in this domain. We then present the SensoByg research project in section 2.2 and the specific requirements identified in that project. In section 2.3, we explain the rationale for choosing the GAF and AMCA routing protocols to be the focus of our investigations.

2.1 Structural Health Monitoring

Structural Health Monitoring (SHM) is a fairly new domain, concerned with monitoring the state of structures. This can include activities such as monitoring the moisture and temperature levels in the concrete that makes up the structure, so that any defects or weak points may be discovered before they cause any serious problems. Even with the most advanced engineering techniques and the best materials available today, all structures decay over time due to exterior forces such as weather effects and seismic events. This means that it is important to monitor the health of any complex structure. Structural health monitoring (SHM) concerns itself with just that and has recently become an important research topic. A typical SHM setup consists of a large structure, e.g., a bridge, and a number of sensors, which are placed on strategic places on or inside the structure. The sensors periodically measure quantities, such as humidity and temperature, which indicate the state of the structure. The data from each sensor is then typically sent to a base station, which acts as a communication central, and often as a coordinator of the sensor network itself. Finally, all the collected data are used to assess the health of the structure.

2.1.1 Sensor Networks in SHM

Until now, SHM have been dependent on wired sensors. Due to the size and complexity of many modern structures, it is a time consuming and expensive task to deploy these sensors. However, recent advances in wireless sensor technology makes it possible to manufacture small, reliable and inexpensive wireless sensor units. This in turn makes it viable to deploy a SHM system on large structures, as a wireless sensor network (WSN).

The shift to wireless technology raises several interesting issues. First and foremost there is the issue of power consumption. As sensor units become wireless, they will have to rely on batteries for power. This causes a problem as it must be assumed that changing these batteries is either impossible due to their placement (i.e., the nodes may be embedded in concrete) or undesired due to the costs of accessing the units and replacing the batteries, versus the cost of a new unit. Hence power consumption becomes the number one issue in a WSN.

To alleviate these problems, many new routing protocols specific to WSNs have been proposed. We describe some of the most prominent ones in section 2.3. Many of these incorporate advanced duty-cycling and take advantage of different low-power modes available in the sensors. The challenge here lies in the fact that it is unlikely that one protocol will work for all scenarios. Therefore many of the proposed protocols focus on specific application domains, and try to maximize the lifetime of the network given the conditions and scenarios within that domain. Some have focused on delivering streams of realtime data while others focus on periodic data gathering and utilizing the energy efficient low-power modes.

These different areas are also seen within SHM. If, e.g. one wants to monitor seismic activity and its effect on a given structure, one would have to sample the seismic activity very frequently in order to determine if there are any disturbances. This scenario is very different compared to, say, monitoring moisture between the inner and outer walls in a building, as moisture penetrates a building at a very slow rate. This means that it is possible to have a very low sample rate and still detect the wanted event. Both of these scenarios are within the area of SHM, but the technologies involved to solve the problems vary significantly.

Of course, energy consumption is not the only factor that must be taken into account when making the transition to wireless technology. The radios of the sensor units have to be limited in size, which means that the radio range is reduced. Combined with the fact that materials such as steel girdles may cause interference to wireless communication means that the network must have a certain amount of fault tolerance, and be able to compensate for environmental effects. Similarly, some sensors may be placed in harsh environments with, e.g. extreme pH values or high humidity. This means that is is very plausible that some nodes will cease functioning unexpectedly due to exterior forces. All these factors must be considered when trying to develop a SHM network in any context.

2.2 SensoByg Project

The SensoByg project[4] is a 3-year research project aimed at embedding cheap wireless sensor networks into small and large concrete constructions. This project is a collaboration between many different companies and institutions in Denmark and is divided into several smaller projects:

- F1 - *System Archtecture*: This subproject is about the overall network and system architecture of the entire system. It will also involve various databases and user interfaces that applications use to access the underlying sensor network.
- F2 - *Decision Support*: This subproject involves creating a system that handles the input and output of the sensor network and all application specific scenarios in the SensoByg project.
- F3 - *Embedding of Sensors*: This subproject is about developing methods for embedding sensors into various construction materials, both during construction or after construction. How and where they should be placed to ensure optimal sensor coverage, and finally to ensure that the number of defect sensor nodes is as small as possible.
- F4 - *Wireless Sensors*: This subproject is about developing wireless sensors that are small, cheap and enable wireless communication.
- F5 - *Coupling and Interaction*: This subproject investigates how various wireless frequencies affect the signal range and how various construction materials affect the signal coming from the sensor nodes.
- D1 - *Moisture in houses*: This subproject is about developing and demonstrating the advantages of using wireless sensors to monitor moisture in houses today.
- D2 - *Measurements in large construction*: This subproject is about using wireless sensors to measure moisture in large constructions (e.g., bridges and large buildings). This will typically involve larger sensor networks then those mentioned in D1.
- D3 - *Maturity of concrete elements*: This subproject is about using wireless sensors to notify when a concrete element has matured enough to allow transport from the factory to the building site where it will be placed.
- D4 - *Moisture in the build phase*: This subproject is about using wireless sensors to notify when a concrete element has matured enough so that in can be used in the construction phase (e.g., when a concrete floor is dry enough to allow some material to be put on top of the concrete floor).

- *SensoByg Framework*: Gather all the results from the above projects and develop that into a business model that can be used in the construction business.

2.2.1 Specification and Requirements

Since we are focusing on routing protocols in large scale SHM, it is irrelevant for us to look into any of the F projects. The D2, however, provides a suitable setting for our routing protocols to be evaluated in. The reason we took that project and not the other D projects is due to the fact that is the only project that would feature large scale wireless sensor networks and is therefore the only project that has nontrivial routing of packets, where trivial routing is when all the sensor units are within one hop of the base station. The other three D projects would be possible to achieve using direct communication between sensor nodes and the base station. Direct communication is not possible in the D2 case since the number of sensor nodes is very high, possibly thousands of nodes in one sensor network.

As for the life time of the network, then D3-4 would have a short life time which is basically the construction time. D1 has a longer life time, but it is very unlikely that the sensors would be embedded into the concrete since they are supposed to measure the moisture in the rooms, not in the walls. This means that it is possible to change the battery in the sensor and thereby prolong the lifetime of them. This is not possible in the D2 case. Here the sensor are embedded into the concrete. They will also have to last 5-50 years, so making the nodes sleep most of the time is a significant concern here. The nodes are also deployed over a large area making direct communication with the base station extremely difficult if the expected network lifetime is supposed to be achieved.

2.2.2 Our Choice of Routing Protocols

The routing protocols needed for our purpose have to possess some specific qualities. Most importantly they have to include some mechanism for allowing most of the nodes in the network to sleep for extended periods of time. GAF and AMCA (which are described in chapter 3) both handles this in a promising manner, yet they differ in their approach. This makes it interesting to see how these protocols will compare to each other.

Secondly the protocols will have to be very scalable. SensoByg concerns itself with monitoring large structures, which requires deployment of an extensive WSN. Hundreds or even thousands of nodes may be needed in order to collect the needed data. The two protocols we have chosen both show promising theoretic scalability. GAF, through its grid overlay which only requires one node in each cell to be active, and AMCA through its dynamic tree structure, that allows for variable cluster heads.

It is also interesting to see how these protocols compare to each other in a setting such as SensoByg, as they represent different types of routing protocols. GAF is mainly location based, although it does contain a hierarchy. AMCA on the other hand, is a pure hierarchical protocol, which does not need any type of location information.

2.3 Routing Protocols

Having a SHM setup with a large WSN requires a way to get the data from the sensors to the base station. Using one-hop communication from the nodes to the base station is not a viable option, as most nodes will not be within one-hop distance of the base station. This is where routing protocols come into the picture. But we cannot use routing protocols that are used on the Internet today. These routing protocols rely on the fact that they can exchange routing data frequently, have excessive storage capacity and have renewable power supplies. Even using routing protocol designed for mobile ad hoc networks is not sufficient[10] since nodes in those kinds of networks usually have more power and storage capacity then wireless sensors. That is why we have to consider routing protocols that are specifically designed for wireless sensor networks. There are many of them available today and since this is an active research field, improvements are being made continually in routing protocols for WSNs. The routing protocols are usually grouped into the following categories: data-centric, hierarchial, and location-based. In this section we briefly describe the basic ideas behind the protocols in each category and give a few examples of typical protocols within each category.

2.3.1 Data-centric Protocols

The idea behind data-centric routing is that the base station sends out a query to either the entire network, or parts of it, requesting data from the sensor units. The base station then waits until the data has propagated all the way back from the sensors.

A very simple routing protocol in this category is the *Flooding* protocol. The idea is that each node broadcasts its message to all its neighboring nodes. If a node receives a message that is not addressed to it, the node will rebroadcast it making sure that the message moves forward in the sensor network. However this approach is not very efficient, as a node might receive the same message multiple times, as shown in figure (2.1). Here we see that a message from node 1 is broadcast to nodes 2 and 3 which rebroadcasts it to node 4. This means that node 4 receives the same message twice. This also means that nodes waste energy broadcasting redundant messages. One way to fix that is to figure out which node should send what data. This is what the protocols, that are described later in this section, do. There is also the possibility that nodes close to each other try to broadcast at the same time, thereby causing collision of packets, which also wastes energy. Therefore, this is not a viable routing protocol for a large scale network.

One of the first non-trivial data-centric routing protocols to emerge was *Sensor Protocols for Information via Negotiation)* (SPIN)[8]. The basic idea behind SPIN is that when a sensor node has new measurement data, the node broadcasts a message saying that it has new data, to its neighbors. Any neighborhood node that does not have the new data will then request the data from the original node. The original node will then broadcast the requested data. An example of this can be seen in figure 2.2 (image redrawn from [8]). One issue with the SPIN protocol occurs when a node interested in the data is far away from the node with the data and the nodes in between are not interested in the data.

The interested node will then never get the data. Therefore is this not a viable solution for our purpose since it is essential that all data is delivered, to all interested parties. In our case study only the base station is interested in the data which would mean that all the intermediate nodes would not deliver data to the base station since the intermediate nodes are not interested in the data.

2.3.2 Hierarchial Protocols

Hierarchial based protocol are popular since many of them scale very well and in some of them, nodes are allowed to sleep to save energy. A hierarchial protocol usually divides the network into clusters where each cluster is a collection of nodes that are close together in a geographic region. Then a clusterhead node is chosen from each cluster to organize all communication between local nodes and clusterhead nodes. This allows local nodes to only transmit their data a short distance to their clusterhead, thereby saving energy.

One of the first hierarchial based routing protocol was *Low-Energy Adaptive Clustering Hierarchy* (LEACH)[7]. The protocol is divided into rounds and each round is divided into two phases: The setup phase and the steady phase. The setup phase, which is the shorter phase, is where the clusterheads are determined and also where it is decided which nodes belong to which cluster. The steady phase is a much longer phase, where measurement data are transmitted from each node to the base station. Randomization is used to determine whether a node should become a clusterhead. A node selects a random number in the range from 0 to 1. If the number is below some given threshold then the node becomes a clusterhead. This threshold is based on the remaining energy, the number of times this node has been clusterhead and the number of suggested clusterhead in the network. This number of suggested clusterheads in the network is calculated and given to each node when the network is initialized. If a node becomes a clusterhead, it will gather data from local cluster nodes and transmit their data, including its own, to the base station in just one hop. A node that is a non-clusterhead only has to transmit data to its clusterhead.

The biggest drawback of LEACH is the fact that all clusterheads must be in a one-hop range of the base station since the clusterheads all communicate directly with the base station. This puts pressure on clusterheads that are far away since they will spend more energy in transmitting their data. There is also alot of overhead in administrating the dy-

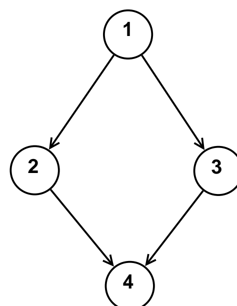


Figure 2.1: Example of flooding.

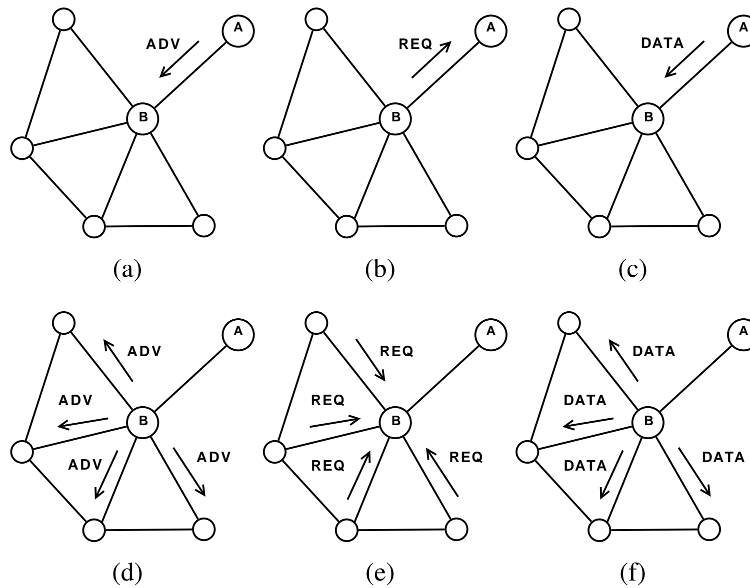


Figure 2.2: Node A sends out an advertisement packet in (a), node B sends a request packet for the data in (b), node A sends the data to B in (c), B sends out advertisement packet to neighborhood nodes in (d), the neighborhood node request the data in (e), and finally B sends out the data in (f).

namic clusters. This means that this protocol is not suited for SensoByg since the sensor nodes will be deployed over a large area and not all nodes will be able to communicate directly with the base station.

Another routing protocol that aims to improve LEACH is *Power-Efficient Gathering in Sensor Information Systems* (PEGASIS)[13]. In PEGASIS a chain is created that forms the routing topology. Each node will be a part of the chain, having two neighborhood nodes, unless the node is an end-node of the chain. In that case it only has one neighborhood node. The chain is created using a greedy algorithm, but it requires that all nodes have global knowledge of the network. This greedy algorithm will choose the node that is closest to it as the next link in the chain. The node that is furthest away starts the chain creation process. One way for a node to know that it is furthest away from the base station is by giving each node a coordinate and since all nodes have global knowledge of the network, they can easily determine which one is furthest away from the base station. An example of a chain can be seen in figure 2.3. In this figure node 0 begins the chain creation process. It chooses node 1 as the next link in the chain. Node 1 chooses node 3 as the next link. Finally, node 3 chooses node 2. If a node dies then the chain creation process must be run again.

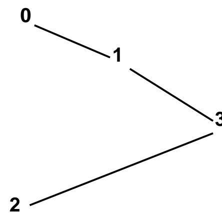


Figure 2.3: Example of a PEGASIS chain.

When the initial chain creation process is done, the routing protocol moves into the data gathering phase. This phase is divided into rounds. In each round a leader is chosen. The leader will send out a token to one end of the chain by sending the token through all intermediate nodes on the chain. The node that received the token will send the token along with its measurement data to the next node on the chain. Here the node will aggregate its own measurement data with the measurement data from the previous node and send that further along the chain. The reason for using a token is so that a node knows when it can send data further along the chain. An example of data gathering in PEGASIS can be seen in figure 2.4. Node C2 is the leader. It sends out the token to C0 along the chain. C0 sends its measurement data to C1 along with the token who then aggregates the data from C0. C1 then sends the token along the data to the leader C2. C2 then sends the token to C4. The process then repeats itself and finally the C2 has gotten the data from C3 and C4. When the leader has gotten data from all nodes, it sends it to the base station in a single hop.

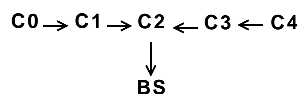


Figure 2.4: Example of data gathering in PEGASIS.

By having a random leader in each round the lifetime of each node is extended. However, each node must be in transmission range of the base station because every node can become the leader. This also means that nodes further away from the base station will waste more energy sending data to the base station compared to nodes closer to the base station. But since there is only one leader in each round of the PEGASIS protocol and there are considerably more leaders in each round in LEACH then PEGASIS does extend the lifetime of the entire network, compared to LEACH.

PEGASIS will not work for SensoByg due to the fact that nodes need to have global knowledge of the network which is not viable in a large scale sensor network. Also if a node dies then the chain creation process has to be run again and that is very expensive in a large scale network. Finally single hop communcations that is required by the leaders is not possible in our SensoByg scenario.

2.3.3 Location-based Protocols

The idea of location based routing protocols is to use knowledge about the location of the nodes to route data. It might be to query a specific geographic region of the sensor network or to route the data the geographically shortest path to the base station, thereby spending less energy in doing so. The nodes are assumed to know their location using either GPS, or some other sort of ranging device to determine their relative position to neighborhood nodes.

One example of a location based routing protocol is *Geographical and Energy Aware Routing* (GEAR)[18]. GEAR uses the fact that nodes know their locations and can save energy by sending the queries to a specific part of the network instead of flooding it. Each node knows its energy status and the energy status of its neighbors. When a node has to forward a packet, it will choose a node that is closer to the destination. If no nodes are closer then the current node then a node further away is chosen based on cost calculations. An example of a route going around a hole in the network topology can be seen in figure 2.5. A hole in the network is when a node has died and is unable to route messages through it. The nodes in figure 2.5 that have died, are colored black. S wants to forward a packet towards T so it chooses C as the next node since that is node closest to T. When C tries to forward the packet there is no node that is closer to T then C. C then has to route the message to one of the nodes that are further away. C chooses B and updates its cost function. B also cannot send to a node that is closer to T then itself therefore it chooses A as the next node in line. A finds a node that is closer and sends the packet to F. This continues until the packet has arrived to T. After a few rounds of routing, S learns from C that C routes to B. S can therefore route the packet directly to B and thereby save energy in the overall network lifetime since C is not routing any data.

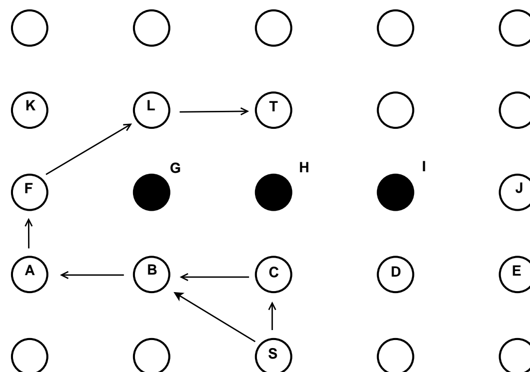


Figure 2.5: Example of data routing in GEAR.

When a packet has arrived at the targeted geographic region by going through multiple hops on the way, it must be delivered to all nodes in that region. The routing protocol specifies two ways of doing that. Either flood that specific region with the packet or use an algorithm called *Recursive Geographic Forwarding*. An example of this algorithm can be seen in figure 2.6. When a node receives a packet that is destined for that region, the node will split up the region into four subregions. A copy of the packet is then sent to all four subregions. There the process is repeated until there is only one node in all subregions.

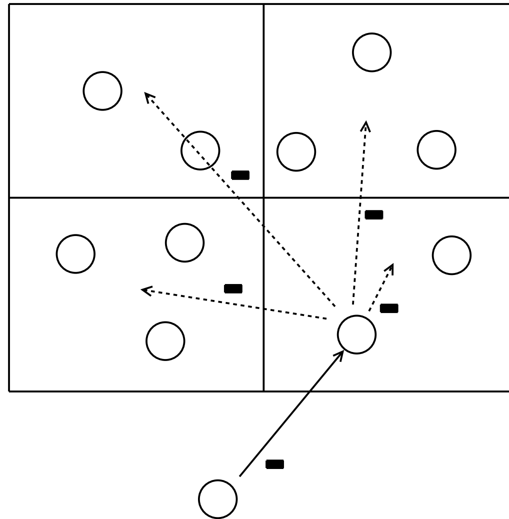


Figure 2.6: Recursive Geographic Forwarding.

This ensures that all nodes receive the packet with only a small amount of energy spent. GEAR is used more for querying specific parts of the network, than sending data back regularly to the base station. This makes it less suited for a SHM scenario, such as the one we are investigating.

The routing protocols we have looked at in this section are by no means the entire available selection of routing protocols for wireless sensor networks. These routing protocols are some of the most representative protocols for wireless sensor networks in each category. We looked at many more during our initial research phase but describing them all is beyond the scope of this thesis. In chapter 3, we describe the routing protocols that we chosen to implement and evaluate in this thesis.

3

Description of Protocols

This chapter describes the protocols that we have chosen to implement and also show how they have been adapted to the SensoByg project. In section 3.1, we describe how the Geographical Adaptive Fidelity routing protocols works. In section 3.2, we describe how the Autonomous Multicast-tree Creation Algorithm works.

3.1 Geographical Adaptive Fidelity

Geographical Adaptive Fidelity (GAF)[16] is an algorithm to reduce energy consumption in ad-hoc wireless networks. It is independent of any underlying routing protocol, and uses application and system level information to achieve its goals. It does this by addressing some key issues in wireless networks:

- Nodes in close proximity will overhear each others radio transmissions. This results in increased energy consumption in the individual nodes, as they have to waste processing power on messages that may not be meant for them.
- Not all nodes are required in order to keep a constant routing fidelity throughout the network. This means that some nodes may be powered off, without losing overall connectivity in the network, in the sense that any node who tries to send a message to another node, will be able to find a route to the receiver at any given time.

What this means is that in order to achieve long lifetimes in wireless networks, it is essential that nodes are powered down when they are not needed. The immediate problem

that arises from this observation, is of course the fact that the network still needs some degree of connectivity in order to function. E.g., if all neighbors of a specific node decide to sleep at the same time, the node will not be able to send any messages through the network, and hence the network has lost its connectivity. GAF handles this problem by identifying nodes which are redundant from a routing perspective. This means that it maintains full connectivity throughout the network, using a minimum amount of nodes.

3.1.1 Grid Overlay

In order to reduce overall power consumption, GAF introduces a virtual grid, which is placed on top of the nodes in the network. The grid is built, such that it is possible for every node within one grid cell, to communicate directly with all awake nodes in the four neighboring grid cells. E.g. node *A* in figure 3.1, must be in one-hop distance to nodes *B*, *C*, *D* and *E*. In order to ensure that this is always true, the grid is made up of squares with size *r* times *r*. Seeing as any given node must be able to reach all nodes in neighboring grid cells, we can conclude that a node must have a minimum radio range of what corresponds to the hypothenuse in the triangle obtained by splitting 2 grid cells put next to each other from one corner to the other, as depicted in figure 3.2. This means that it is possible to compute *r*, from the expected average radio range, *R*, of the sensor units in the network. Using Pythagoras's theorem, we get the following equation [16]:

$$r^2 + (2r)^2 \leq R^2 \tag{3.1}$$

Which can be rewritten to:

$$r \leq \frac{R}{\sqrt{5}} \tag{3.2}$$

The clever thing about the grid overlay is that only one node in each grid cell is required to be active in order to achieve a constant routing fidelity, as illustrated in figure 3.1. This means that as soon as a grid leader has been established in a grid cell for some period of time, the other nodes in that grid cell can enter sleep mode for that period,

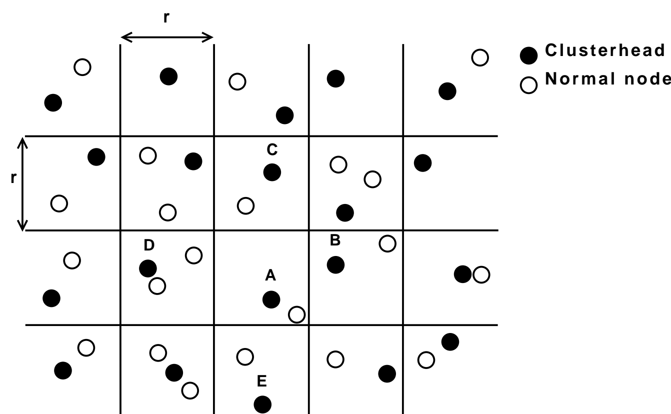


Figure 3.1: Virtual grid in GAF.

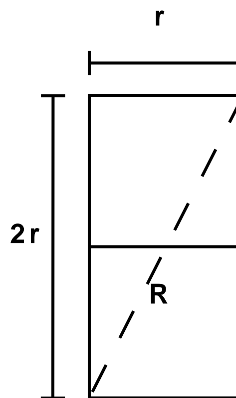


Figure 3.2: GAF grid size and expected radio range R

without worrying about breaking the connectivity of the network. In theory this means that the overall lifetime of the network benefits from greater node density, as the positive effects of having redundant nodes turned off will increase as more nodes reside within each grid cell.

Given that a node knows *R*, its own location and the location of the base station, it is possible to determine locally which grid cell the node belongs to. This means that no central initialization or calculations are required in order to build the grid. As a direct consequence of this, it has to be assumed that each node is aware of its own location. Either in relation to some fixed point (e.g. the base station), or in some global context (e.g. via GPS or other means). Similarly the nodes have to be aware of their expected average radio range. This value can be determined given that the hardware platform of the nodes are known. Alternatively initial experiments may have to be performed to measure the radio range in the context of the deployment site.

3.1.2 Node States

Each node alternates between 3 states: Sleeping, discovery and active, as illustrated in figure 3.3. When nodes are in the sleeping state, all unnecessary circuits such as the radio will be turned off, to conserve as much power as possible. Only after a specific timer *Ts* expires, will the node wake up and enter the discovery state.

In the discovery state the node will first set a timer *Td* and then assemble a discovery message consisting of the following items:

- Node id.
- Grid id. Determined using the location of the node and the grid size.
- Estimated node active time (*enat*). Estimated time that this node can be expected to be active, once it enters the active state.
- Estimated node life time (*enlt*).
- An indication of whether the node is leader of the grid cell.

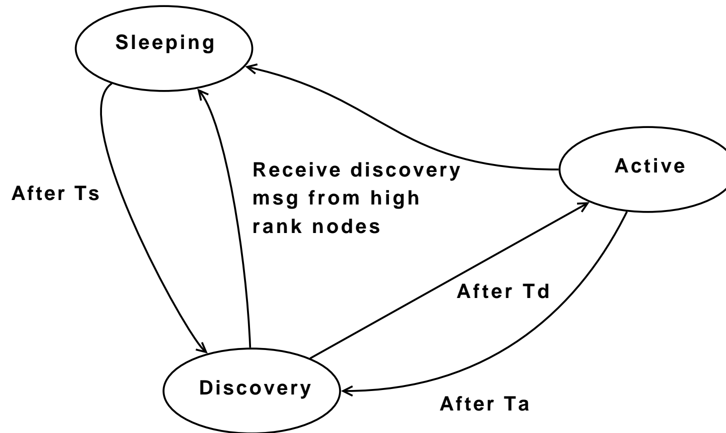


Figure 3.3: GAF node state diagram.

When the timer *Td* expires, the node broadcasts the discovery message and enters the active state. *Td* is typically chosen at random in some interval, to ensure that nodes do not broadcast messages simultaneously. The purpose of the discovery message is to communicate node status to neighboring nodes.

When the active state is entered another timer *Ta* is set. When this timer expires, the node will have served its time as leader and will reenter the discovery state. While in the active state, the node will continuously rebroadcast the discovery message at some set interval. Note that when a node is in either the discovery or the active state, it can at any time transition to the sleeping state again. This happens as soon as it is determined that a node will handle routing in the grid, e.g. when the new grid leader sends a discovery message announcing that it has become leader. Hence, the discovery state exists so that it is possible for nodes to make informed decisions as to whom should handle routing. Locally, a node will decide whether it should handle routing based on the discovery messages it receives from its neighboring nodes. The local node will calculate a *node rank* for each node, and the highest ranking node will then be considered responsible for routing all traffic from within the grid cell, to the correct destinations. By having the same ranking algorithm distributed on all nodes, it is possible to make this selection locally at each node, instead of having a global selection algorithm in place. This process is made simpler because of the structure of the grid overlay, explained earlier, where each node in a given grid cell is ensured to be able to communicate directly with all other live nodes in the four neighboring cells.

3.1.3 Parameters and Node Ranking

There are a number of parameters involved in the GAF algorithm. We have briefly touched upon some of them already, but in this section we will explain each parameter in more detail. Firstly though, we will introduce the concept of *node ranking.*

Node rank is used as a way for higher ranking nodes to suppress messages from lower ranking nodes. E.g. if a high ranked node and a low ranked node simultaneously decided that they would handle routing in the grid, then the higher ranked node would suppress the

lower ranked node, and start routing messages. There are several different things which influences node rank. Most importantly, an active node has higher rank than a node in the discovery state. As there is only one active node in a grid at any time, this rule enables a larger degree of control for that node. Secondly, a higher *enat* value (see below for definition) will grant a higher rank. This helps balance load onto nodes with higher energy levels. Finally, node identities will break any ties which may occur.

The algorithm to calculate node rank is run locally on each node based on the discovery messages which is received from its neighbors. Hence, if a node sends out a discovery message, all neighboring nodes will calculate the same rank for the node. As stated earlier, this knowledge is used to select which nodes should route messages in each grid cell, and also works as a mechanism for higher ranking nodes to take priority over lower ranking ones.
We will now explain the different parameters in the GAF algorithm.

- *enlt* - Expected node lifetime. Time before the node runs out of power.
- *enat* - Expected node active time. A measure of how long the node can be expected to be active in the current active cycle. Usually calculated as some fraction of *enlt*.
- *Td* - Timer used to determine when the discovery message will be broadcast. Usually a random value in some interval [0, *k*]. This timer can be influenced by node rank. By allowing higher ranking nodes to have a very short timer, they can allow lower ranking nodes to enter the sleep state faster as they will be able to determine that they should not be responsible for routing. When a node enters active mode, it is usually a good idea to set *Td* to a higher value, so that the discovery message is not broadcast too often.
- *Ta* - Timer used to determine when a node will exit the active state and enter the discovery state. In order to balance the load in the network, it is important that *Ta* is less than *enlt*. In the default GAF algorithm *Ta* is set to *enat* = (*enlt* / 2) when entering the discovery state. If *enlt* is less than some threshold value, *Ta* is set to the remaining lifetime of the node so that thrashing is avoided, where several nodes with low energy could continually exchange leadership in rapid succession until they run out of energy.
- *Ts* - Timer which decides when a node will wake up from its sleep state. By default GAF picks this value at random in the interval [*enat* / 2, *enat*].

3.1.4 Routing in GAF

As stated earlier, GAF is not a routing protocol in itself. Rather, it is an overlay network which will work with any underlying routing protocol. In our experiments we will use simple greedy geographic routing. The GAF overlay makes this approach efficient and easy to handle, and it requires no extra information other than that which is already exchanged between nodes in the the GAF algorithm. This is based on the connectivity

guarantees that GAF supplies, which makes greedy geographic routing straightforward. It is even further simplified by thinking of the network as a grid. This way we only have to choose between the four neighbor grid cells when routing a packet to a destination. In fact, we only have two concrete choices if we are using a greedy geographic routing protocol. Namely the two that will get the message closer to the target grid cell.

Doing this in a simple manner, however, may introduce a number of problems. The most immediate one is load balance. If we always deterministically choose the shortest path to route packets, then we will overload certain nodes which is on the most direct path to the destination. This would make some grid cells deplete their energy much faster than the rest of the network. What we want is a simple way to distribute the workload across the network.

The way we will address this problem is by using the local information each node already has about its neighboring nodes and grid cells. Specifically the energy level is a key factor here. By choosing to route packets toward grid cells with a higher energy level, we will balance the load *locally*. This scheme will not result in the most energy efficient route from a global viewpoint, as it will still always route packets closer to its destination at every intermediate node. However, this approach requires no extra information or state, and hence does not introduce any overhead. And given the structure of our overlay network, it should still improve upon the straight forward greedy routing approach.

3.2 Autonomous Multicast-tree Creation Algorithm

The *Autonomous Multicast-tree Creation Algorithm* (AMCA)[15] protocol creates a tree structure which it then uses to maintain the network and route data packets from nodes to the *sink*, which then sends the data directly to the base station. There is no need for an explicit routing protocol since each node communicates with either its parent or one of its children. To route data to the base station each node will communicate with its parent and that way the data is forwarded up the tree. To extend the lifetime of the network the algorithm proposes that some of the nodes sleep while others are active to maintain full connectivity in the tree. To overcome the problem of losing a subtree whenever a parent dies, the algorithm proposes that each node maintains a list of *clusterhead* candidates. This clusterhead candidate list contains all the nodes that are potential clusterheads for each cluster. A clusterhead is a node that acts as a parent for another node lower in the tree. Whenever the child has to send data to the sink, the child will forward the data to the clusterhead, who will then forward the data further up the tree, until it finally arrives at the sink. There, the sink sends the data directly to the base station.

The algorithm works in cycles. In each cycle the nodes maintain the tree structure and make sure that there is full connectivity in the network. In figure 3.4, we see an example of a network layout after the initial tree structure has been made. Here the nodes have been divided into clusters and all data is propagated up the tree.

Table 3.1 contains the information that each node stores. This information is sent out to neighboring nodes to create the initial tree structure and to maintain it afterwards.

Variable name	Description
ID	Node id number
IDsink	Current sink node id
IDprev	Previous sink node id
L	Depth in the tree from the sink node
G	The number of times the root has changed
E	Remaining energy of the node
Cadopt	The number of children that this node has
P[n]	A list of clusterhead candidates
pktType	Type of packet being sent

Table 3.1: Information stored in nodes.

3.2.1 Initial Tree Creation

After the nodes have been deployed in the field, they enter a tree creation phase, where they collectively build the tree structure used in this routing protocol. One of the assumptions made in the routing protocol is that each node has been given a specific id. Either this is done by giving each node an id manually during deployment or have the nodes determine themselves which id they should have. Either way is possible and we will not go into details on how this is done. The ids should be arranged so that the further away from the base station, a given node is, the higher id it should have. The node with id 0, will become the sink in the tree. This will form the basis for the initial tree creation phase.

At initial deployment there is no tree structure in place that the nodes are part of. So the first thing that each node has to do is to determine the sink and the level of the tree it is in. This is done by having each node send out a control package that contains the information in table 3.1 to the neighboring nodes. To make a node change its sink, one of the following three conditions must be met:

- The node receives a message from a neighborhood node saying that the sink node has been replaced. This can happen if the previous sink node has lost connectivity or is out of power. This means that a new node has become the sink and that infor-

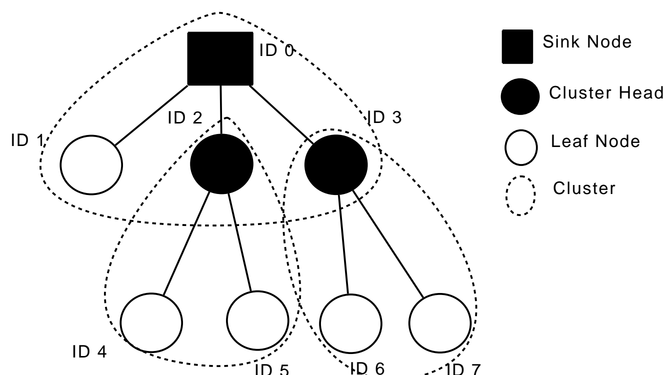


Figure 3.4: Example of network layout.

mation has to be propagated to all nodes in the tree.

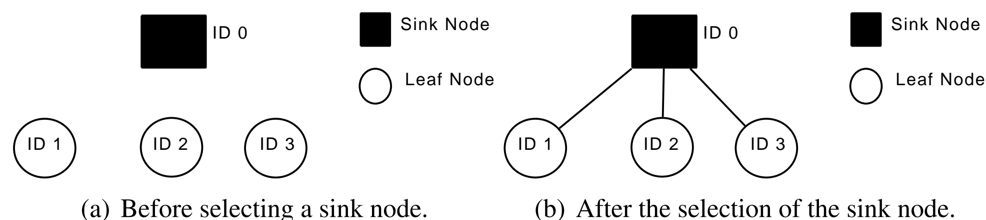
- The node receives a message from a neighborhood node that states the number of sink changes are less then the number of sink changes this node has done.

- The id number of the sink in the message is less then the id number of the sink in the node. This is the condition that initially starts the tree creation process. An example of this can be found in figure 3.5. In figure 3.5(a) node 0 has not sent out any control messages. In figure 3.5(b) node 0 has sent out a control message which has resulted in nodes 1-3 having registered node 0 as their new sink node.

Accordingly, to make a node change its level, the following condition must be met:

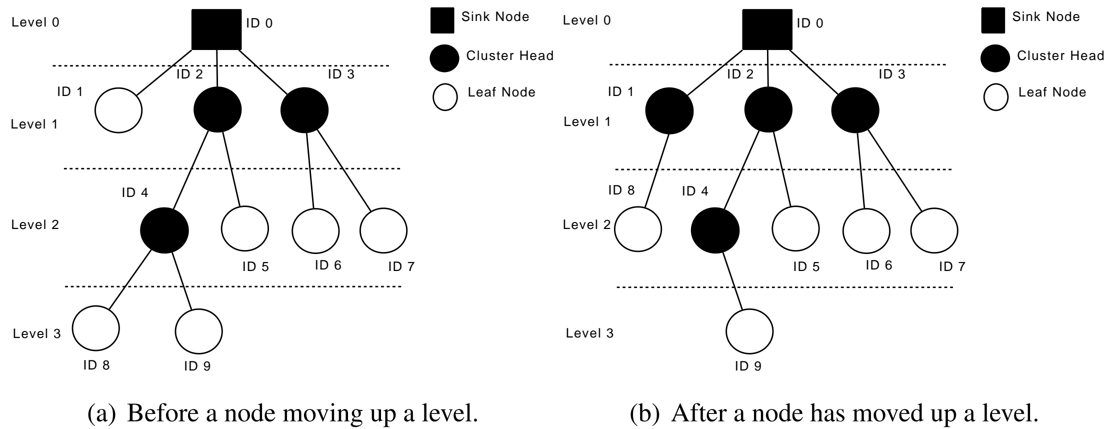
- A node receives a message from a neighboring node which states that the id of the sink, in the message, is the same id that the current node is storing but it resides on a level that is closer to the sink node then the current node. This might allow the node to go up one or more levels in the tree structure. An example of this can be seen in figure 3.6. In figure 3.6(a) node with id = 8 is in the subtree rooted at node with id = 4. However, in figure 3.6(b), node 8 receives a message from node 1 allowing it to move up one level.

To ensure that all nodes do not send packets at the same time, which would mean that they would block out each others signal, a delay has been introduced into every layer of the tree structure. This delay is a constant integer of the value $0 < t1 << T$ where T is the cycle time that each node operates in and $t1$ is the level delay. This T is the same for all nodes in the network. The sink node has no delay time, level 1 has delay of size t1, level 2 has delay of size $2 * \mathrm{t1}$ and so forth. This can be seen in figure 3.7. Each node sends two messages. One message in the beginning of each cycle and another one at some interval into the cycle. The later message is called the *broadcast message*. The message in the beginning of the cycle, is basically an announcement saying that this node is alive. Then in the broadcast message, a node has decided who will be its clusterhead. After a few rounds exchanging this information there should be a fully connected tree structure in place. Should it happen that a node is not connected in the initial tree construction phase



(a) Before selecting a sink node. (b) After the selection of the sink node.

Figure 3.5: Selection of new sink node.



(a) Before a node moving up a level. (b) After a node has moved up a level.

Figure 3.6: Example of a node moving up a level.

the node can always connect to the tree later on in the tree maintenance phase. After the tree has been created the algorithm moves onto tree maintenance.

3.2.2 Tree Maintenance

When the tree is under construction then *T*, which is the time for each cycle in the algorithm, should be a relatively small value to ensure that the tree is created fairly fast. But when the tree moves into maintenance mode then *T* should be given a value that is much larger. This is to ensure that control packets are not being sent all the time.

There are two things that each node has to do in the maintenance phase: Choose a clusterhead and decide which children it adopts. A node may choose between several clusterhead candidates. This is because a clusterhead can run out of power or become unreachable for some reason. This would mean that the entire subtree of that clusterhead node would become unreachable. This would be catastrophic if it were to happen high up in the tree. To alleviate that problem, the algorithm proposes clusterhead candidates. If a

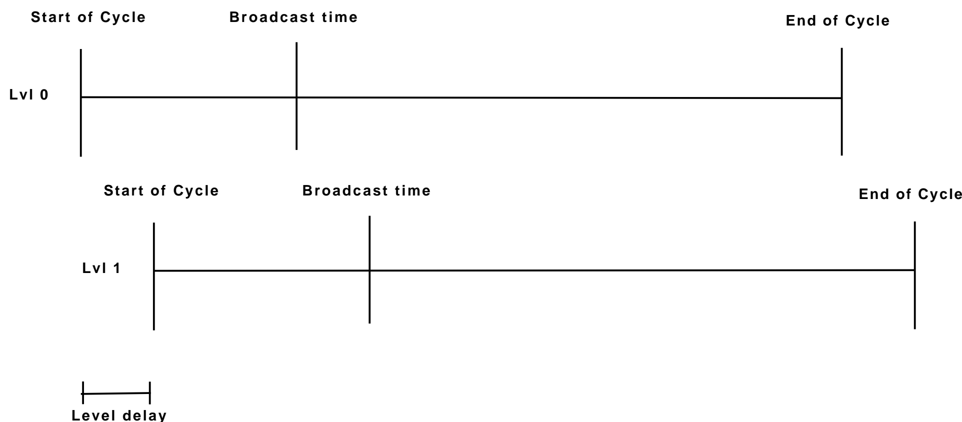


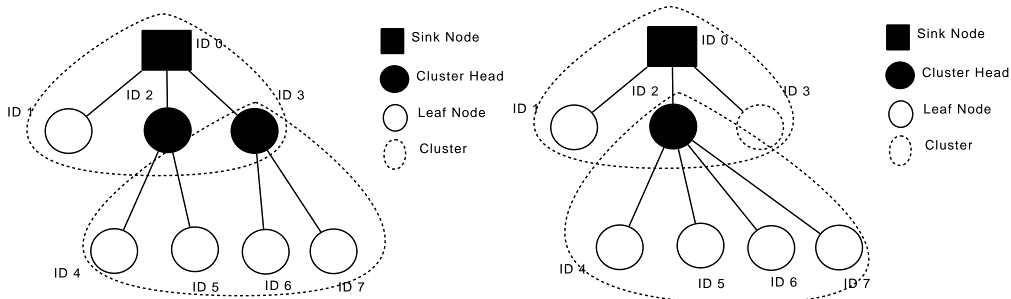
Figure 3.7: Timeline of a cycle.

clusterhead becomes unreachable then the second node on that clusterhead candidate list becomes the new clusterhead. The way that each node maintains all this information is by sending two control messages in each cycle. An example of a recovery after a node has become unreachable, can be seen in figure 3.8. The node with id 3, in figure 3.8(a), has just become unreachable which means that nodes 6 and 7 have lost connection. Since they have a list of clusterhead candidates then they will just choose a new clusterhead. That turns out to be node 2 which then adopts nodes 6 and 7 as can be seen in figure 3.8(b).

One problem that remains in this algorithm is if a subtree becomes disconnected, like in figure 3.9, where nodes have no clusterhead candidates list to fall back on if their clusterhead becomes unreachable. If a node has not received acknowledgment packets from its parent, the node will then mark itself as the sink node. When the next control messages are received from the neighboring nodes, there is a possibility that one of the neighboring nodes can become a parent for the disconnected subtree. This can only happen if the id of the neighboring nodes is lower then the id of the top node in the subtree. In figure 3.9(a), node 3 has become unreachable and nodes 6 and 7 cannot reach any other node in the layer above. Since every node sends out two control messages in every cycle then nodes 6 and 7 will receive a control message from node 5 and thereby make it their parent. This is what has happened in figure 3.9(b). This, however, might take a few cycles to occur but that is at least better than considering the entire subtree lost forever. If node 4 had lost all clusterhead candidates there would be no way for it to be connected again unless the initialization phase would run again.

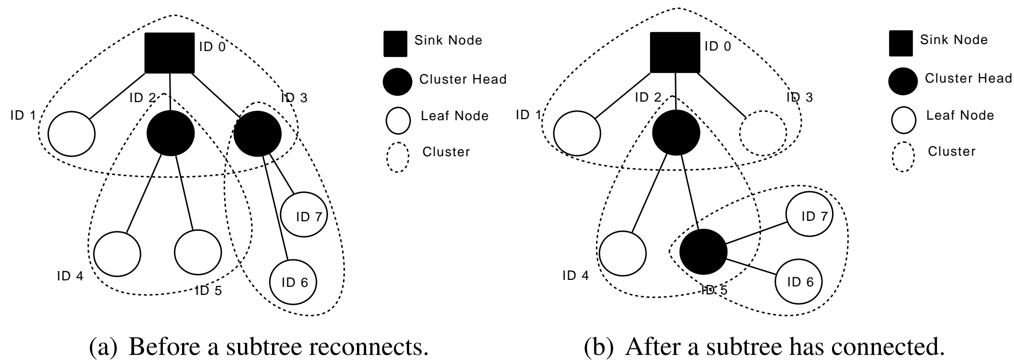
From the tree construction phase, each node knows which neighboring nodes it can reach. Therefore it also knows which nodes are clusterhead candidates. Each node will choose the node with the most remaining energy as its clusterhead as the clusterhead candidate list is ordered by remaining energy. In the first control message each node sends out this clusterhead candidate list, its remaining energy along with the rest of the information in table 3.1. The number of children it has is set to zero in the first message. This is because the node does not know yet which nodes from one level below will choose it to be their clusterhead but it will know that after it has received the first control message from nodes one level below. To see if a node should adopt nodes from one layer below is based on whether these underlying nodes have chosen the current node as their clusterhead.

The final thing that each node must do, is to decide whether the node should stay awake or go to sleep for the rest of the cycle. If a node is on the clusterhead candidate list then it will not sleep. If it is not on that list then it will sleep.



(a) Before a node chooses a new clusterhead. (b) After a node has chosen a new clusterhead.

Figure 3.8: Example of a node choosing a new clusterhead.



(a) Before a subtree reconnects. (b) After a subtree has connected.

Figure 3.9: Example of a subtree being connected again.

One way to increase the number of nodes that sleep is to limit the number of possible clusterhead candidates for each node. This will also make the routing message smaller since the id's of clusterhead candidates is sent in that routing message, thereby saving energy. This will, however, increase the burden on some clusterhead nodes, since this will make some clusterheads adapt more children. This means that they will drain energy faster due to more children sending data up to them. Another drawback of limiting the size of the clusterhead candidate list, is that a node has fewer possibilities to route its messages up the tree if any of its clusterhead candidates become unreachable. This would have to be evaluated on a per case basis. If packets are not lost too frequently then the routing protocol could use a very small clusterhead candidate list since there would not be much use for redundancy. If however the packet loss was high, then a larger list would be needed.

4

Implementation Platform

This chapter describes the environment we used to implement and evaluate our protocols. Section 4.1 describes how programming is done in the TinyOS environment, which we used to implement the routing protocols under investigation. Section 4.2 describes TOSSIM, which is a simulator capable of simulating TinyOS applications. TOSSIM was used to evaluate the protocols. Section 4.2 describes the power consumption model we implemented and used in our simulations. Section 4.4 contains a description of the framework we designed, in order to have a common setting in which we can evaluate the protocols. Finally, in section 4.5, we describe the implementation of our protocols.

4.1 TinyOS 2

When developing software in a WSN context, it is important to address the many limitations imposed by the hardware platforms used in the network. In particular, there is often severe memory constraints which must be carefully considered when doing any kind of implementation. This means that it is usually not viable to use a lot of high level programming abstractions, as they will introduce an overhead in program size. As an example, it is not possible to have a virtual machine running, or make use of large dynamic link libraries.
TinyOS[5] is an open-source operating system designed for WSNs, which tries to address these issues.

4.1.1 Interfaces and Components

TinyOS applications are written in a dialect of the C programming language called *nesC*. They are made up of *interfaces* and *components*. Interfaces specify *commands* and *events*, which components then *provides*. If a component provides an interface, it must include an implementation of all the commands specified in the interface. In addition, the component may also *signal* any of the events from the interface.

There are two different kinds of components: *modules* and *configurations*. Modules are implementations. This is where functions are defined and state is allocated. Configurations on the other hands can wire several components together, into one combined functional component. Thus it is possible to build a new component by using functionality already implemented in other components. As a configuration is in itself also a component, it is possible for a configuration to wire other configurations together with modules. The exact details on how the wiring mechanism works, is discussed later in this section. The following is an example interface, taking from the system architecture we implemented.

```
interface ComSend {
  command error_t sendMsg(msg_route *msg);
  event void sendMsgDone(msg_route *msg, error_t error);
}
```

Here we specify an interface named `ComSend`, which has one command `sendMsg` and one event `sendMsgDone`. This interface can then be provided by a component, as `MessageSenderC`, in the following manner:

```
module MessageSenderC {
  provides {
    interface ComSend;
  }
}
implementation {
  command error_t ComSend.sendMsg(msg_route *msg) {
    //implementation that will send the message occurs here
    return SUCCESS;
  }

}
```

The above example shows the general structure of a TinyOS module component. There is a module definition part, where all interfaces that are either provided or used, are listed. As we can see, the `MessageSenderC` module provides the `ComSend` interface. This means that `MessageSenderC` has to provide an implementation of the command `sendMsg`. As we can see, this is done in the *implementation* part of the module.

Notice that the definition of the command `sendMsg` is prefixed by the interface name `ComSend`. This is due to the fact that nesC does not allow dynamic command invocation.

All calls to commands and all signaling of events, must be statically wired to a specific component. This is why the special *configuration* components are needed, as they wire all interfaces that a module uses, to specific components that provide those interfaces. Before we discuss configurations any further, we will give an example of using an interface and calling a command on this. The following snippet of code is an example module, which uses the `ComSend` component from earlier.

```
module ExampleC {
  uses {
    ComSend;
  }
}
implementation {
    msg_route *myMessage;

    void someLocalMethod() {
     error_t status = call ComSend.sendMsg(myMessage);
    }

    event void ComSend.sendMsgDone(msg_route *msg,
                                  error_t status) {
     // The message 'msg' has now been sent.
    }
}
```

As can be seen from the example, all interfaces which we wish to use, must be specified in the *uses* part of the module. When a component declares that it is using a interface, it is then free to call the commands specified in it. This is done with the *call* keyword followed by the full signature and parameters of the command. In the example above we call the `sendMsg` command, from our `ComSend` interface.

In addition, a component must also declare event handlers for all events specified in the interface it uses. In this case, the `ComSend` interface declares a `sendMsgDone` event, and thus an appropriate event handler has been implemented in our `ExampleC` module. Again, this ensures that there is always a static relationship between events being signaled and the event handlers that receive that event.

Lastly it is worth noting that TinyOS makes excessive use of callback events. This is due to the fact that TinyOS works off a single stack, and is completely nonblocking. If it was not possible to delay computation of invoked commands, it would not be possible to maintain the high level of concurrency that TinyOS does. Hence the preferred way of interacting with an interface that has to produce any sort of result is via events and callbacks. This makes for a more complex program logic as the programmer has to deal with many events that may be internally dependent, but it is necessary in order to provide a reasonable performance.

4.1.2 Configurations

As mentioned in the previous section, the nesC compiler requires a static program structure. This means that all command invocations must be wired to a specific component that provides that command at compile time. Similarly, the compiler needs to know which event handlers will be notified of a signaled event. To achieve this, TinyOS defines a special *configuration* component. The responsibility of the configuration is to *wire* all interfaces used by a module to a specific component which provides that interface. It is also possible for a configuration itself to provide interfaces, as it is also a component. Hence configurations can be seen as a collection of components wired together to provide some combined functionality.

The following is an example configuration which wires the `ExampleC` and the `MessageSenderC` components into a new TinyOS application named `ExampleAppC`.

```
configuration ExampleAppC {

}
implementation {
  components MessageSenderC, ExampleC;
  ExampleC.Sender -> MessageSenderC;
}
```

As a configuration is in itself a component it consists of the same two parts as a module. In the first `configuration` part, it is able to declare any interfaces that it provides. Our example configuration does not provide any interface and therefore this part is left empty. The implementation part is were the actual *wiring* takes place. First all the components that needs to be wired together is declared. The configuration then uses the `->` operand to wire the `Sender` interface, that the `ExampleC` component uses, to the `Sender` interface that the `MessageSenderC` component provides. Basically it is telling TinyOS, that whenever `ExampleC` calls a command from the `Sender` interface, it is calling the specific command implemented by the `MessageSenderC` component. This insures that all command invocations can be statically determined on compile time.

4.2 TOSSIM

The TinyOS Simulator[12](TOSSIM) was created to ease testing and debugging of TinyOS applications. It allows developers to verify and evaluate TinyOS applications, without having to setup large and expensive testbeds with multiple sensor nodes. This eases the task of implementing applications, and helps keep development costs down.

TOSSIM can simulate entire TinyOS applications. All the hardware specific library calls in a TinyOS application, are replaced with TOSSIM implementations of the given hardware components. It can simulate hardware interrupts and high level system events, like receiving packets.

TOSSIM is a discrete event simulator. It maintains a queue of events sorted by the time they are supposed to occur. The simulator will forward the time to when the first event in the queue is scheduled to take place and then simulate that event. TOSSIM does not simulate the time it takes to perform calculations. Therefore events that include very time consuming calculations, will not impact the time.

4.2.1 Running an Application in TOSSIM

TOSSIM can be run in two ways. In the interactive mode, the user has to advance each event manually. This mode allows the user to inspect the values of variables during the simulation. This is mainly used for debugging purposes. The second option is for the user to specify a python file that contains the entire simulation script. This is how most simulations are performed. An example of such a script can be seen here:

```
#! /usr/bin/python
from TOSSIM import *
import sys

t = Tossim([])
r = t.radio()
```

This is the beginning of the file. Here the necessary libraries are imported and a TOSSIM object and a radio objects are created.

```
f = open("topo.txt", "r")
lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    r.add(int(s[0]), int(s[1]), float(s[2]))
```

Here the topology file is read in and the values from it are added to the radio object. This creates the radio links between the nodes. The specifics of the topology file is described later in this section.

```
t.addChannel("RadioCountToLedsC", sys.stdout)
t.addChannel("Boot", sys.stdout)
```

This creates channels that can be used for debug statements in the nesC code. This is explained better in section 4.2.2.

```
noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
  str = line.strip()
  if (str != ""):
    val = int(str)
    for i in range(1, 4):
      t.getNode(i).addNoiseTraceReading(val)
```

This part reads in the noise trace file and adds that information to each node in the network topology. The noise trace file that we used, comes with the TinyOS distribution. The file contains hundreds of thousands of noise trace values taken from a real-life network. They are used to simulate the amount of noise that occurs on the radio channels over time.

Using the full version of this file, requires around 10 megabytes of memory per node in the simulations. So simulating very large networks requires computers with vast amounts of memory. Since we did not have such computers at our disposal, we had to settle for using the first 1/8th of the noise trace readings. This means that the accuracy of the simulated noise is decreased, but this does not affect our simulations in any noticeable way, as the amount of readings we use are more than adequate for our purposes.

```
for i in range(1, 4):
  print "Creating noise model for ",i;
  t.getNode(i).createNoiseModel()
```

This creates the noise model for each node. The noise model is based on the noise trace readings from before, and is used to determine the noise on the radio channel when any given packet is transmitted. Thus, influencing if the packet is lost.

```
t.getNode(1).bootAtTime(100001);
t.getNode(2).bootAtTime(800008);
t.getNode(3).bootAtTime(1800009);

for i in range(0, 100):
  t.runNextEvent()
```

This final part of the example file, starts the nodes at different times and then runs through 100 events in the simulation. To create the network topology, we have to create a topology file that has the following format:

```
source, destination, gain
```

Source and destination creates a one way link from the node specified in the source, to the node specified in the destination. The `gain` specifies the receivers strength of the signal when a message is sent across the link. The greater the strength, the more likely the packet will delivered successfully. If the packet is successfully received or not, depends on the strength of the signal and the noise on the communication channel according to the noise

model. An example topology file can be seen here:

```
0  1 -54.0
1  0 -55.0
1  2 -55.0
2  1 -57.0
```

This file describes the network topology where node 1 has bi-directional links with node 0 and node 2. Node 0 and node 2 have no links between and hence they cannot communicate. There is no notion of distance in TOSSIM. Saying that there is a link between node 0 and 1, says nothing how of far they are apart. As the routing protocols we implement are dependent on distances, we created a topology generator that alleviates this problem. This topology generator creates a grid of size x * y. Then it assigns ids into each cell of the grid. Finally it loops through all cells in the array and determines which nodes are in radio range of each other, based on the range that is given as a parameter when running the program. To find the distance between two given nodes, we use the common distance formula for points in a two dimensional plane:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{4.1}$$

where x and y are the coordinates for the two different grids. An example of this is in figure 4.1. Here, the distance is set to 2 meters, and thus all nodes that are within two grid cells distance to node 5, will have a bidirectional link to node 5. The red line in the figure is the two meter distance from node 5. In this example, nodes 1, 3, 7, 15, 4, 6 and 16, have a link to node 5. This is done for all cells in the grid, thereby creation a notion of a distance in our simulations.

4.2.2 Using Debug Statements in TOSSIM

Running the simulator and not seeing any output whatsoever is of no use to TinyOS programmers. That is why it is possible to write debug statements into the nesC code. The debug statements have similar syntax as *sprintf* statements from C++.

```
dbg(channel name, string to print out, a list of variables);
```

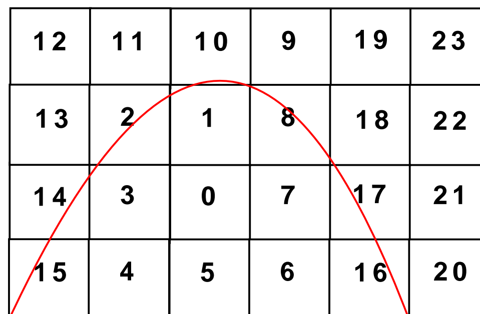


Figure 4.1: Example of a topology.

The following is taken from our nesC code as an example of such a debug statement :

```
dbg("Cadopt", "Number of children adopted: %u \n", Cadopt);
```

This will print out the string along with the value of Cadopt into the channel called Cadopt.

But before we can get output from the simulator, we have to specify in the python script, a channel that is supposed to print out the string. That is done in the following way:

```
t.addChannel("Cadopt", sys.stdout)
```

This will add a channel to TOSSIM called Cadopt and it will print out the statements to the console. It is also possible to write to a file instead of `stdout`. As each debug statement writes to a specific channel, it is just a matter of selecting the appropriate channel in the python script if we want the output from it. If more advanced debugging is needed, it is possible to use a driver written in C++ instead of using a python script, as we have seen above. Doing so allows the use of common debuggers such as *gdb*.

4.2.3 Our Experience with TOSSIM

Our experience with the TOSSIM simulator, is one with mixed emotions. It is very convenient to able to run our code in the simulator without doing any modifications to it. However, we noticed that it runs very slowly when doing simulations with many nodes and where events happen very infrequently. In theory, TOSSIM should be able to support around 1000 nodes but our experience shows that it is impractical when simulating large networks over a long time period.

We decided to do a small experiment to see how well TOSSIM scales. In figure 4.2, we can see what happens if either the number of simulated nodes is increased or the time on the timers, that the nodes use, is increased. The red line indicates how much the time increases when we increase the nodes from 1 to 500. If we double the number of nodes, then the duration of the simulation is also doubled. If we double the time on the timers, then the duration of the simulations is also doubled. This means that doubling the number of nodes and doubling the time on the timers, quadruples the simulation time.

We also noticed that the simulations seemed to slow down as we got further into them. We have no explanation why that happened. When running the simulations we saw no indications that there was any memory leak in our program since the memory usage stays constant during the simulations. We also had simulations that we had to terminate because it was not possible for them to finish in any realistic timeframe.

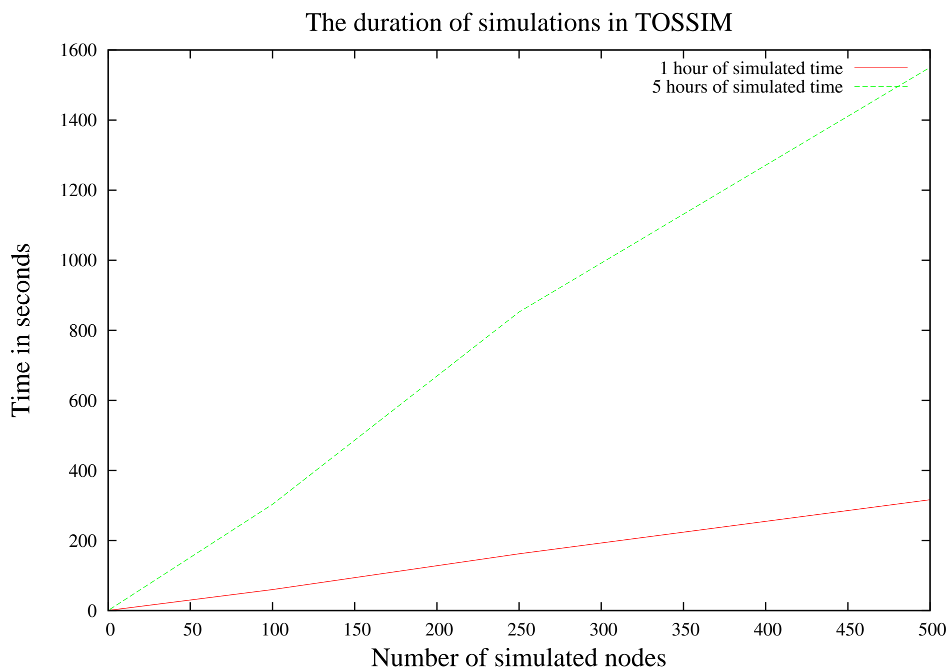


Figure 4.2: Running times of TOSSIM simulations.

4.3 Power Simulation

Due to the lack of available simulators that simulate energy consumption for TinyOS applications, we decided to implement our own model for simulating energy usage in our sensor nodes. It is a fairly simple model which we now describe in more detail.

4.3.1 Energy Calculations

To be able to create a useful power simulation, we have to know the initial energy that each node starts with and then how much energy each node spends on different activities. To estimate the energy used, we use the data sheet from the MICAz[3] node platform. This is the hardware platform we are compiling our simulations for, and is one of the most widely used platforms for this purpose.

Initial Energy

The MICAz is designed to carry two AA batteries[2]. An AA battery has a nominal voltage of 1.5V[1] and 3.000 mAH[2]. This gives us a total of 3.0V and 6.000mAh or 6 Ah.

[1]V stands for voltage
[2]mAh is milliAmpere-hours and Ah is Ampere-hours

All equations used in this section are common energy calculations[17]. First we need to calculate the amount of watt-hours present in a battery:

$$P = I * V = 3.0V * 6.0Ah = 18Wh \tag{4.2}$$

We now need to convert the output from equation (4.2) into joules[3] since that is the common unit in both energy used and energy stored. So we multiply the outcome from equation (4.2) with 3600, which is the number of seconds per hour.

$$18Wh * 3600s = 64.800J(Ws) = 64.800.000mJ \tag{4.3}$$

Equation (4.3) gives us 64.800.000 mJ as the total number of energy that each node can spend.

Energy spent on sending/receiving messages

To figure out how much power each node uses on the radio when receiving and sending messages, we first have to know how much time it takes to send and receive one bit over the radio. The MICAz node has a 250kbps radio[3], so that gives us the following:

$$t = \frac{1s}{250.000bits/second} = 0.000004\text{s per bit} \tag{4.4}$$

Since the MICAz radio uses a different amount of energy based on whether it is receiving or sending messages, we need to calculate the energy spent in separate cases. Let us start with, when the radio is sending a message. The MICAz radio has three different power settings for the radio when sending messages. The higher the power setting, the stronger the signal is sent out, making it more likely to be received by other nodes in the network. We decided to use the one that uses the most power in sending messages since in the SensoByg project, the idea is that the nodes communicate with each other through concrete. This will of course lessen the lifetime of the network but should give us a better idea of worst case lifetime of the network. So using the value of 17.4 mA in sending the message, we see that the radio uses the following power in sending messages:

$$P = I * V = 17.4mA * 3.0V = 52.2mW \tag{4.5}$$

The equation used is the same equation as in (4.2). The final thing we need to do in order to determine the energy spent per bit, is to multiply the outcome from (4.5) with the outcome from (4.4).

$$E = P * t = 52.2mW * 0.000004s = 0.0002088mJ \tag{4.6}$$

So, 0.0002088mJ is the amount of energy needed to send one bit with the maximum signal strength on the MICAz radio.

As for the energy spent receiving one bit, we go through the same steps as the ones used to calculate the energy spent sending one bit. The only difference is the current used in the radio. In sending one bit, the radio used 17.4mA while receiving it uses 19.7mA. This gives us a total of 0.0002364mJ of energy used per bit received.

[3] One Joule is one Watt per second

Energy spent on CPU and idle radio

Here we need to know how much energy a node uses when the CPU is active. Since it is not possible for us in this simple energy model to count CPU cycles, we assume that when the node is awake, the CPU is in active mode. In real-life situations, this would not always be correct, but again, this will give us a worst case estimate of the lifetime of the node.
Similarly, we always assume that the radio is on when the node is not sleeping. The radio spends 0.060mW when idle in this manner. Again, it is possible to have the node active while the radio is off, but in the simulations we are doing, the node will only be active if it wants to use the radio. Therefore, this simplification will not affect our results.

What remains to be calculated, is how much energy is used when the node is sleeping. Here, both the radio and the CPU are sleeping. The calculations are the same as when calculating the energy used for a node in the active state. The final results can be seen in table 4.1

State	Energy spent
Radio send	0.0002088mJ per bit
Radio receive	0.0002364mJ per bit
Active mode(CPU + radio)	24.060mJ per second
Sleep mode (CPU + radio)	0.048mJ per second

Table 4.1: Energy usage of various modes.

4.4 System Architecture

In order to accurately compare our two routing protocols performance against each other, we have created a common routing layer framework. The framework constitutes a complete routing layer for a node, which can be used directly by the application layer. It is responsible for handling all messages that are being sent and received and it provides the needed buffers and queues for these purposes. The framework provides a *RouteSelector* interface, that each of our routing protocols implement. A component implementing this interface, is responsible for determining the next-hop receiver of any application level packet being sent or routed, and for inserting any routing specific information that might be needed. The *RouteSelector* component also has direct access to the communication module of the framework and can send out control messages at will. Any control message a node receives will be directed to the *RouteSelector* components from the framework.
In the next sections we take a look at the individual parts of the framework and describe it in more detail.

4.4.1 Overview

The framework consists of five components as depicted in the static relationship diagram in figure 4.3. An elaboration on the notation used, can be seen in figure 4.4. The components are:

- Router - Main component that controls all other components in the framework. Responsible for handling all message reception and distribution.
- MessageSender - Component used for sending application level messages. Also provides and maintains buffers for these messages to the application layer.
- RouteSelector - Responsible for assigning routes to messages being sent and handling all routing control messages.
- Communication - Provides a general communication abstraction. Responsible for maintaining a queue of messages being sent, and for providing and maintaining message buffers to other components.
- EnergyModel - Responsible for maintaining an estimate of the nodes energy level. The energy is influenced by the amount of time spent in active or sleeping mode, and the amount of messages sent and received, including the size of these messages.

A custom message format has also been specified, which are intended for use with application layer components. The routing application is aware of how to route these messages, and all other types of messages are treated as routing control messages. The message format *msg_route* is specified as so:

```
typedef nx_struct msg_route {
  nx_uint32_t msgID;
  nx_uint16_t destinationNodeID;
  nx_uint16_t senderNodeID;
  nx_uint16_t routedFromNodeID;
  nx_uint8_t data[16];
} msg_route;
```

The *msgID* is a unique integer used for identification of messages. The *routedFromNodeID* specifies the node which we received the message from. Only the *data* and *destinationNodeID* needs to be set by the application sending the message. The other fields are handled by the routing component.

4.4.2 Router Component

The *Router* component is the entry-point for applications who want to use the routing layer. It is responsible for starting and stopping the other components in the framework

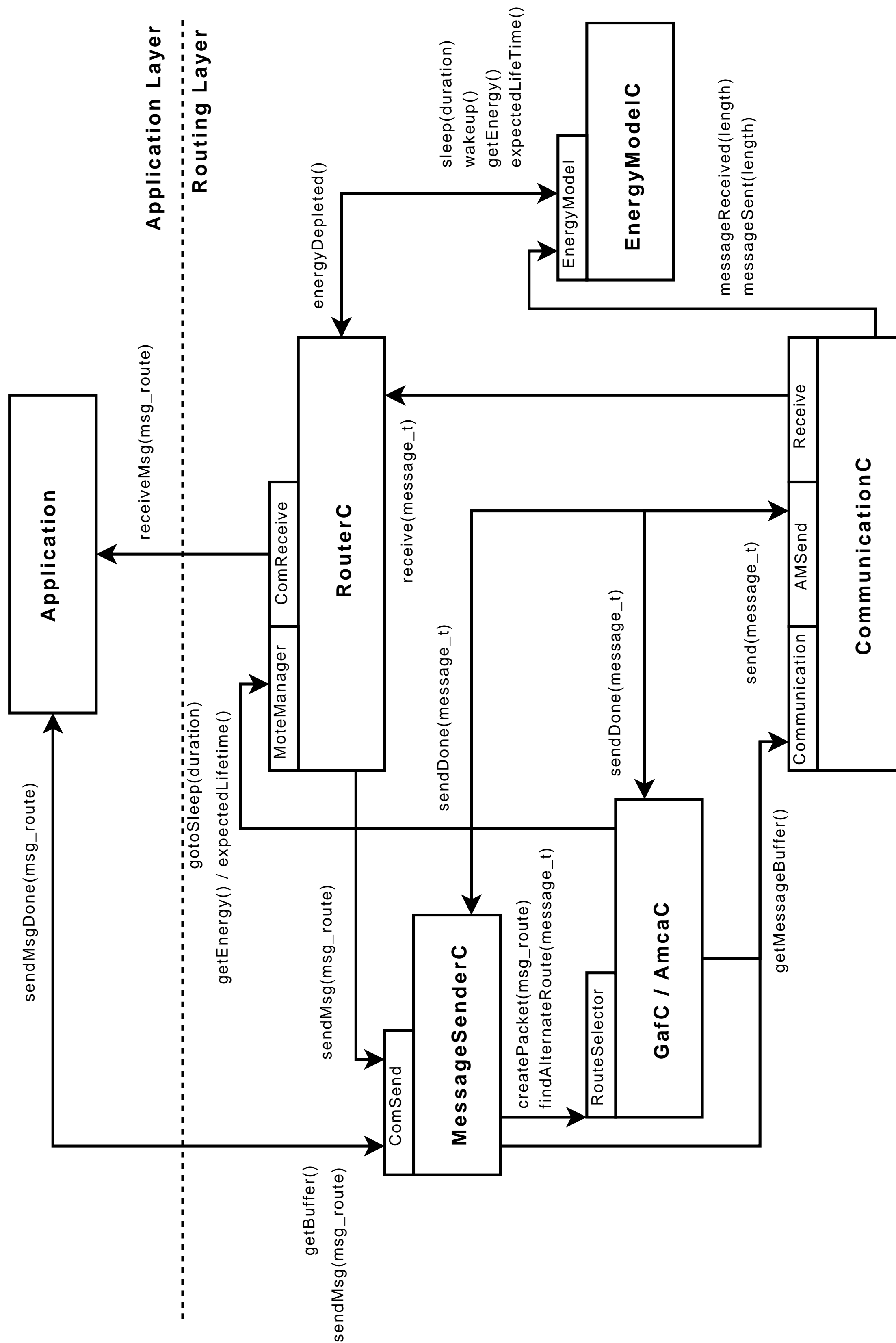


Figure 4.3: Static overview of the framework.

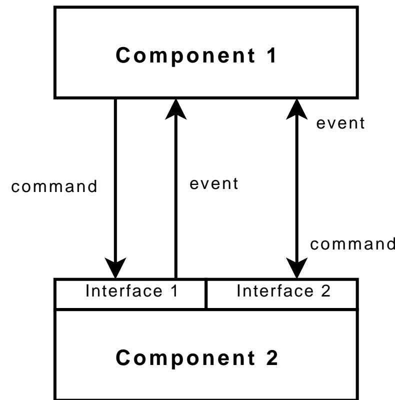


Figure 4.4: Notation used in the framework description.

via the *SplitControl* interface. This interface is provided by the other components meaning that they provide a *start()* and a *stop()* method for controlling the state of the component.

The Router component is also responsible for handling all incoming messages. It does this by wiring to the *receive* event provided by the *Communication* component. All received messages are classified by the type of payload they carry. If it is a *msg_route*, then the destination will be checked to see if it is a message for the current node. If not, then the message will be routed along using the *MessageSender* component. In case it is a message for the current node, the message will be delivered to the application layer via the *ComReceive* interface which the Router component provides. This interface specifies a single event: *receiveMsg(msg_route *msg)*

If the payload is not a *msg_route*, then it is assumed that it is some kind of routing specific message. Hence, it is directed to the *RouteSelector* component via the *process-RoutingPacket* command, which the RouteSelector interface specifies.

Should further message types be introduced, it is easy to extend the receive method to detect this type, and redirect it to the correct component.

Furthermore, the Router component provides the *MoteManager* interface, which contains the *gotoSleep* command. This command will make the node sleep for the specified amount of milliseconds. In order to put the node to sleep, and wake it up accordingly, the *start* and *stop* commands, which all components provide via the *SplitControl* interface, are used appropriately.

The MoteManager interface also contains several commands concerning the nodes energy level. It is possible to get the remaining energy in millijoule or percent. It also provides a command to get an estimate of the remaining node lifetime in seconds. These commands are mostly pass-through calls to corresponding commands in the *EnergyModel* component. They are provided in the MoteManager interface, as to hide the functionality pertaining to consuming power from the *RouteSelector* implementations. Hence any routing algorithm can access the nodes energy level, but is unaware of the underlying energy calculations going on in the framework.

4.4.3 MessageSender Component

The MessageSender component has two responsibilities: To provide the functionality to send *msg_route* messages, and to provide *msg_route* buffers to other components. The functionality to send *msg_route* messages is provided via the *sendMsg* command in the *ComSend* interface, which is used by the application layer and the Router component. The interface also specifies a *sendMsgDone* event which is fired when the message has been delivered to the outgoing message queue, and a *getBuffer* command which is used to retrieve *msg_route* buffers. Any buffer that has been allocated, will automatically be released when the message is sent via the *sendMsg* command.

The process of sending a *msg_route* message with the MessageSender component, is illustrated in the sequence diagram in figure 4.5. After a message is passed to the *sendMsg* command, it will be given a unique message id and the *routedFromNodeID* field will be assigned the id of the current node. A *message_t* buffer will then be allocated from the *Communication* component, and the message and the buffer is then passed to the *RouteSelector* component using the *createPacket* command. Here the message to be send will be copied into the buffers payload section. The *RouteSelector* will then determine the node id of the next hop node the message has to be routed to, and assign this value to the buffers destination field. If a next hop address was successfully determined, the buffer will then be delivered from the *MessageSender* component to the outgoing message queue in the *Communication* component.

After the message has been sent over the radio, the *MessageSender* component will receive a *sendDone* event. If the transmission was successful, a *sendMsgDone* event will be signaled to the sender of the message. If it was unsuccessful, a retry mechanism is in place, to improve the robustness of the framework. A *msg_route* will be retried 3 times. If it has not been sent successfully at this point, the *RouteSelector* component will be asked to find an alternate route to the destination. If this is possible, the message will be sent again. This goes on until the message is sent to the destination, or it is not possible to find any more alternative routes.

4.4.4 RouteSelector Component

The *RouteSelector* component has the responsibility of implementing a routing algorithm. Hence it provides functionality to create packets that needs to be routed, and to process any incoming routing messages. This is where our GAF and AMCA implementations are situated. Typically a routing algorithm will need to send out control messages, which is done via the Communication component. This ensures that the RouteSelector component shares the same message buffers as the rest of the framework.

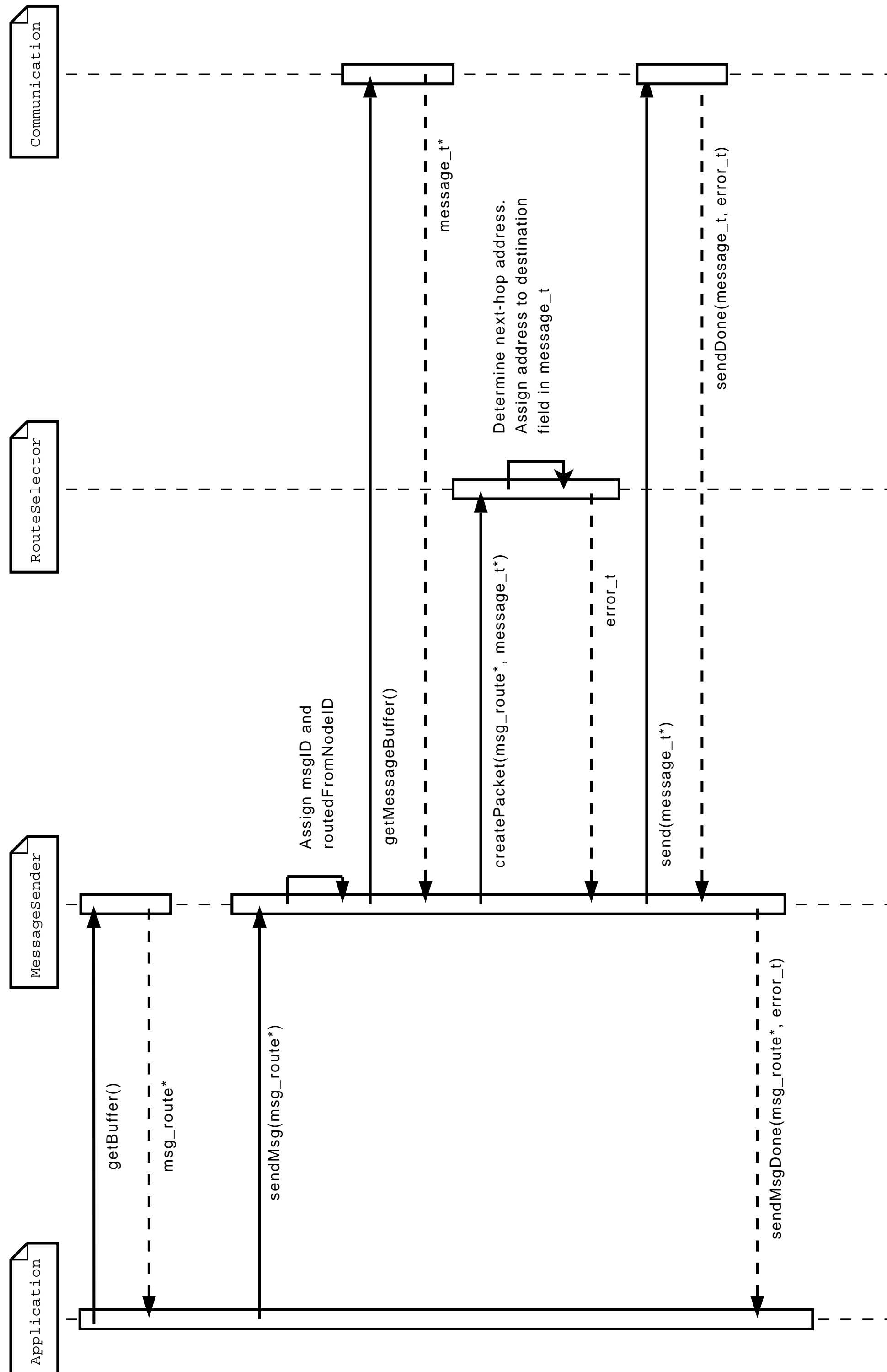


Figure 4.5: Sequence of events happening when a message is sent successfully

4.4.5 Communication Component

The Communication component works as the generic access point to the radio. It provides all *message_t* used via the *getMessageBuffer()* command in the *Communication* interface. Buffers are automatically released and reused as the requested buffers get sent off.

One of the important aspects about having an abstraction for the radio, is the implementation of a queue for outgoing messages. This relieves other components of the bothersome task of continually monitoring if the radio is busy. TinyOS comes with a generic *Queue* component which is used for this purpose.

All message reception also goes through the Communication component. When a message is received it is copied to an available message buffer and the *receive* event is signaled. In our case, the Router component receives and handles all such events.

As mentioned earlier in this section, there is a retry mechanism in place in the *MessageSender* component, to improve the robustness of the framework. In order for this to work, the Communication component has to be able to determine when a message has not been delivered to the recipient. To achieve this, it utilizes the packet acknowledgments that are built into TinyOS. As it works on the hardware level of each node, it is transparent to the framework. All that is needed is to set the *ack* field in a *message_t* to let the radio know that the message should be acknowledged. If a node receives a message which requires an acknowledgment, an *ack* packet will be sent in response. However, as these packets are subject to the same radio interference as any other messages, they may be lost as well. This can lead to situations where a node will report that a message was not delivered, even though it was actually received by the correct recipient. This goes to show that even though an acknowledgment protocol is in place, our retry mechanism can not lead to any one-hop delivery guarantees. Lastly, it is the responsibility of the Communication component to invoke the appropriate commands in the EnergyModel, whenever a message has been sent or received.

4.4.6 EnergyModel Component

The *EnergyModel* component is responsible for maintaining an estimate of the nodes current energy level. It achieves this by providing four commands which are called from the Router and Communication components. The commands utilizes the energy calculations presented in section 4.3.1, to calculate the amount of energy spent over time when in either the active or sleeping mode. The amount of energy spent when sending or receiving a given amount of bits, will also be calculated. The commands involved will now be explained in more detail.

```
command void messageSent(uint8_t len)
```

This command is called every time a message has been sent over the radio. This happens whether the message was actually delivered or not, as the amount of energy consumed is the same in both cases. The *len* parameter describes the length of the message sent in bytes.

```
command void messageReceived(uint8_t len)
```

This command is called every time a message is received. Again, the *len* parameter describes the length of the message received in bytes.

```
command void sleep(uint32_t duration)
```

This command is called whenever a node goes to sleep. The *duration* parameter is needed, as the energy model will have to calculate if the node will run out of energy while it is sleeping. If this is the case, a timer is set to fire at the time when this happens, so that the node may be notified that it has run out of energy.

```
command void wakeup()
```

This command is called whenever a node exits the sleeping state. The energy model needs to know when the node is either awake or asleep, so that the energy calculations can take this into account. Internally, a time stamp is maintained in order to keep track of how much time has gone by between each energy update.

The EnergyModel component also contains commands to retrieve the energy and the estimated node life time in seconds. The latter is calculated purely based on the amount of energy used in either the sleeping or active state, as indicated by a parameter. Again the reader is referred to section 4.3.1 for details. The commands in question are:

```
command double getEnergy()
command double getMaxEnergy()
command uint32_t expectedNodeLifetime(bool activeMode)
```

Lastly, the *EnergyModel* contains an event which is signaled when the node has run out of energy:

```
event void energyDepleted()
```

4.5 Implementation of the Routing Protocols

While implementing the two routing protocols under investigation, we were faced with certain implementation specific decisions. Largely these issues arose from the memory constraints apparent on our hardware platform, but some optimization for better energy efficiency has also been done. In this section we touch upon the choices we made in that respect, and comment on the reasons behind them.

4.5.1 Time Synchronization

In our implementation it is assumed that all nodes have synchronized clocks. This is a simplification of the real world, where clocks will drift over longer periods of time. However, none of the protocols require millisecond precision, and will function as long as the clocks does not drift more than a few seconds from each other. Several time protocols have been proposed and tested, which should be able to achieve this kind of precision, however it is beyond the scope of this thesis to implement and test a time synchronization protocol on top of our framework. However, given that such a protocol was chosen, it would be straight forward to plug it into the framework we have created.

Furthermore, given that such a protocol can function independently of the routing layer, the overhead induced would be independent of the routing protocols we are testing. Thus, given that it is possible to synchronize nodes with a certain precision, our results will still be valid. Two examples of such time synchronization protocols are *Rate-adaptive time synchronization for long-lived sensor networks*[6] and *The Flooding Time Synchronization Protocol*[14]. Both protocols are able to synchronize the clocks of the nodes within a few milliseconds range.

4.5.2 Base Station

In all respects, the base station is just a regular node, running the same software as all the other nodes in the network. To be able to distinguish the base station, we have decided that it is always the node with identity zero that takes on this role. Furthermore it is assumed that the base station has access to an external energy supply, such that it will actually be able to receive the messages being sent to it, throughout the lifetime of the network.

4.5.3 GAF

The implementation of the GAF protocol, can be found in the *GafC* TinyOS component. As described earlier, this component provides the *RouteSelector* interface, so that it is able to provide route selection for the framework. A couple of tweaks have been made to the protocol described in section 3.1 in an attempt to make it more energy efficient. As mentioned, in the default implementation a node that is grid leader, will periodically broadcast its discovery message, to inform neighboring nodes of its status. However, as other nodes in the grid will go to sleep as soon as they determine that a leader is active, there will rarely be any interested nodes who will receive this discovery message. As sending unnecessary messages results in increased energy consumption, we have tried to alleviate this problem.

A node that is leader will expect to be replaced sometime in the last k seconds of its active time. k is a simulation parameter is experimentally decided. As the estimated node active time (*enat*) for a leader is announced in its discovery message, this allows all the other nodes in the grid cell to sleep for *enat* seconds minus the threshold value k. When

the nodes wake up, the grid leader will be ready to replaced, and the discovery timer (*Td*) for the recently awakened node is chosen at random in the interval [k/2, k]. This ensures that a new leader will be selected before the current one expires. The randomness in the selection of *Td* was introduced to prevent all nodes from sending their discovery message at the same time, thus blocking the radio channel for each other.

During the period where the nodes are in the discovery state, the current leader will broadcast its discovery message every *k/4* seconds. This is to ensure that all nodes have a high probability of receiving the message despite the risk of packet loss. Likewise, when a new leader becomes active, it will send out its discovery message with the same interval for *k* amount of seconds.

Should the unlikely event happen that a node's discovery timer expires without the node having received a message from a leader, it will assume that there is no leader active and become leader itself. This may lead to two leaders being active in a given grid cell. To alleviate this problem a *leaderTimeStamp* field has been added to the discovery message. This is used when a leader node is not ready to be replaced, but it receives a discovery message from another node claiming to be leader as well. In this case the leader with the earliest time stamp will take priority, and the other node will give up its leadership and go to sleep. The final discovery message used in the implementation, is represented as follows:

```
typedef nx_struct msg_gaf_dsc {
    nx_uint16_t nodeID;
    nx_uint32_t leaderTimeStamp;
    nx_uint8_t isLeader;
    loc gridPos;
    nx_uint32_t enat;
    nx_uint32_t enlt;
} msg_gaf_dsc;
```

Loc is a structure defined for convenience:

```
typedef nx_struct loc {
  nx_int16_t x;
  nx_int16_t y;
}loc;
```

This results in a total message size of 19 bytes. As TinyOS does not have any way of serializing booleans, 7 of these bits are wasted as we have to represent the *isLeader* field as an 8 bit unsigned integer.

As our primary interest is extending the lifetime of the network, there is no formal node ranking in place as described in section 3.1. Instead each node will decide if they should replace the current leader, solely based on the estimated time to live (*enlt*) announced in the discovery messages. The only other form of node ranking is when two leaders are active in a grid cell, where the earliest leader will take priority as described above.

In order to conserve as much energy as possible, we are also interested in getting

the nodes to sleep as fast as possible. Thus, a node in the discovery state will not wait for its discovery timer to fire before going to sleep, if it has received a message from a new leader. Neither will it broadcast its own discovery message, as there will be no nodes which have any use for it. In practice this means that the first node that has their discovery timer fire and has more energy than the current leader, will become leader. As soon as the other nodes in the grid cell receives the discovery message from the new leader, they will go to sleep for the announced *enat* time minus *k*. This means that the only discovery messages sent during a grid cells discovery period, is the messages from the old and the new leader, further reducing the energy consumption. The randomized choice for the discovery timer ensures that it is not always the same node that has its discovery timer fire as the first one.

The last thing which must be considered is the timing between neighboring grid cells. If a node becomes leader in a grid cell it will route messages for all other nodes within its own grid. In order to do this it needs to know the identities of the leaders of the four neighboring grid cells. To make this possible, a grid cell leader will also broadcast its discovery message, in the same periodic manner as described above, when one of the neighboring cells enter its discovery period. Again, this time can be determined by the discovery messages received by the neighboring grid cells. Knowing that all nodes are awake during the discovery period, will ensure that all the nodes in the neighboring cell will receive it with a high probability. Therefore, as soon as a node becomes leader it will immediately be able to start routing messages, as it will be aware of all the neighboring leaders as well.

4.5.4 AMCA

All the implementation that is specific for the AMCA protocol, can be found in the following files: *AmcaC.nc* and *amca.h*. The routing message sent between the nodes, is defined as the following struct:

```
typedef nx_struct AmcaControlMsg{
  nx_int16_t ID;
  nx_int16_t IDSink;
  nx_int16_t IDPrev;
  nx_int8_t L;
  nx_uint32_t integerPart;
  nx_uint32_t fractionPart;
  nx_uint8_t Cadopt;
  nx_int16_t P[MAX_NUMBER_OF_ELEMENTS]; //an array of ID's
  } AmcaControlMsg;
```

The size of this message is 16 bytes, plus the number of elements in the P array. This number depends on how many nodes, a single node can choose as its clusterhead candidates. Limiting the number of clusterhead candidates will lead to more nodes becoming leafs, as described in section 3.2.2. We removed *G* from the message since the base station is

part of the main network topology and G stands for the number of sink changes. The sink will never change in our implementation of the routing protocol, as it will always be the base station, which will never run out of energy.

Since nesC does not support any data types for booleans and doubles which can be serialized and sent over the network, we had to send the energy variable, which is defined as a double in local usage, as two 32 bit integers over the network. One containing the integer part and the other containing the fraction part.

As discussed in section 3.2.2, T, which is the length of a cycle, must have different values dependent on whether the algorithm is in the creation or maintenance phase. In the SensoByg setting then T should have a large when in the tree maintenance phase to ensure that the lifetime of the network can be extended. The values of T will be decided in our simulation phase. During the initial tree creation, T is set to 250 seconds, in order to quickly build the tree.

One issue that we had to deal with, was when the nodes send out their routing messages. As seen in figure 3.7, a delay is added between levels so that all the nodes do not send out their messages at the same time. This level delay works well in theory but we had tens or hundreds of nodes sending out at the same time. This would mean that sometimes they would cancel out each others messages. To fix this problem, we introduced a message send interval. This allows each node to randomly send out its message in that interval. The only thing we had to take care of, was that all the messages from a given level, had to be sent before messages from the level below were transmitted. Figure 4.6 shows how these intervals are placed in the timeline of a cycle.

As depicted in figure 4.7, a node will decide if it should sleep or stay awake, shortly after the nodes in the level below have sent their broadcast messages. More precisely, it waits twice the length of the level delay (as explained in section 3.2.1) after having sent its own broadcast message, as this ensure that it will have been notified if any of the nodes

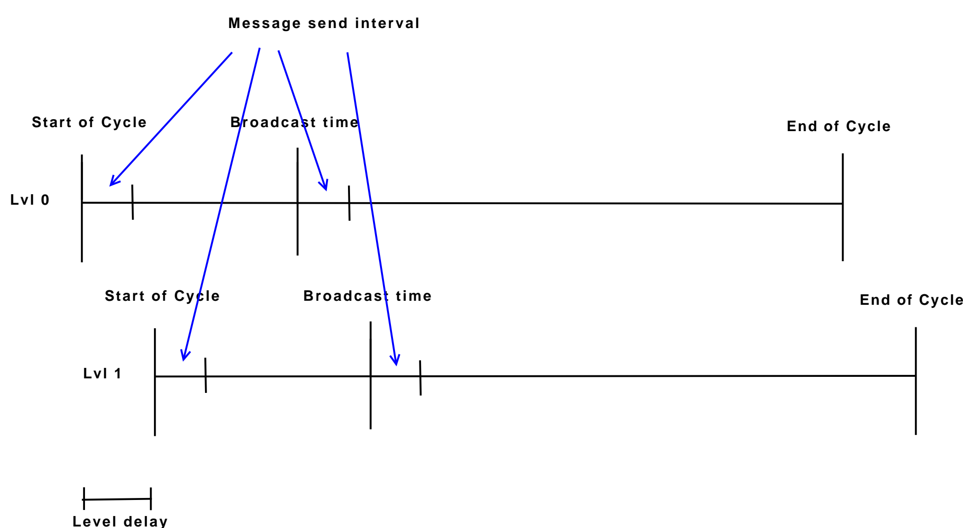


Figure 4.6: Timeline of a cycle with message send intervals.

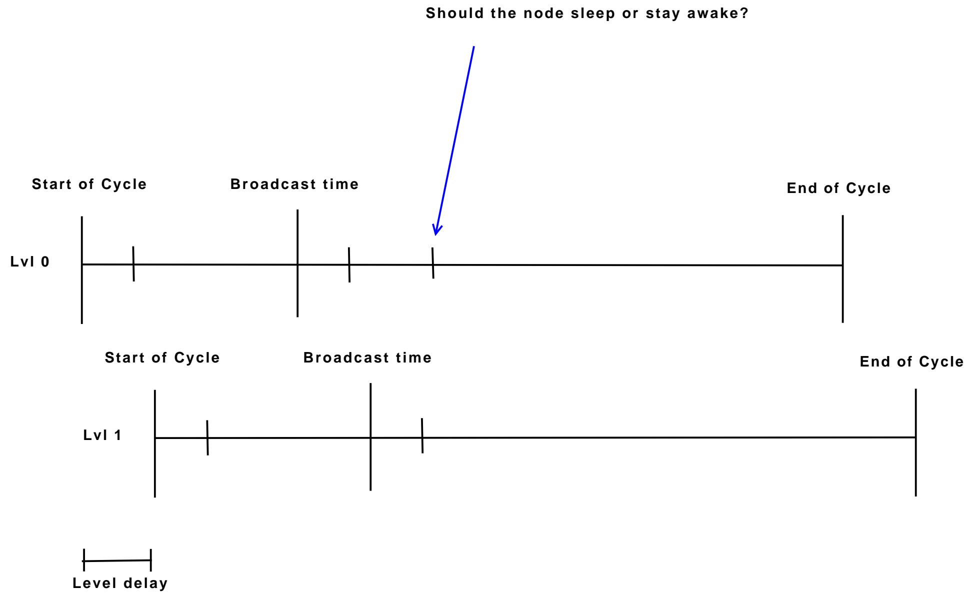


Figure 4.7: Timeline of when a node decides when to sleep or stay awake.

below it have chosen it as a clusterhead or a clusterhead candidate. If it has not been chosen by any nodes, it can enter sleep mode.

Since we assume that the base station has a renewable power supply, then it will never run out of power. This means that one of the variables used in the control message, that is sent between nodes, is unnecessary. This variable is the one that stores the previous base station, called IDPrev in the code. We have, however, included it and all the necessary logic to handle the changing of sink nodes, in case we change the base station into a normal sink. This would mean that the sink node would be a normal node in the network topology and therefore energy constrained. That sink would then at some point become unreachable, meaning that a new sink would have to chosen.

5

Experimental Setup

In this chapter we describe our experimental setup along with the experiments we use to test our implementation of the two protocols. Section 5.1 covers the setup of our simulations and the parameters involved. It also describes the topologies we use and how they relate to the Sensobyg project. Section 5.2 describes the data we extract from our simulations, which is used for performance evaluation and comparison of the protocols.

5.1 Setup

As mentioned in section 4.3.1, we are using the MICAz hardware platform as the basis for our simulations. At the beginning of our simulations, each node has an amount of energy, corresponding to one AA battery. This yields a life-span of just under 16 days for each of the nodes, if they are left in active mode without sending or receiving any messages. This will serve as the baseline for our experiments. The goal is therefore to extend the lifetime of the network as far as possible beyond this baseline.

As a measure of performance for the protocols, each node will send one routing message to the base station, every 24 hours. We will track each message and record the amount of messages that are lost before reaching the base station. This amount will be one of our metrics for comparison. This is to simulate the fact that each node will be collecting a sample from any built-in sensors and sending the data collected to the base station of the SHM network. We have chosen a relatively infrequent sample rate of 24 hours, as this is in trend with the SensoByg studies, where we are collecting humidity and temperature readings from within the concrete building elements that make up a bridge. As moisture will only penetrate concrete at a very slow rate, it is not necessary to collect

readings very frequently, as the readings will not fluctuate much.

In order to alleviate the problem of all nodes sending their data to the base station at the exact same time, a delay has been introduced in the test application. This is necessary because of the time synchronization in TOSSIM, which would mean that most of the messages would be lost because of the wireless communication channel being blocked by all the nodes sending at once. The delay is based on the identity of the nodes, so that node 1 will broadcast its message first, followed by node 2 and so on. The delay between the broadcasts has been set to five seconds, to ensure that the node before has had ample time to send its message.

Despite the delay, nodes may still experience that messages are lost. Partly due to the noise on the radio channel, and partly because data messages may still interfere with each other as they get routed closer to the sink. However, this is expected by the framework and the built-in retransmission mechanism described in section 4.4 ensure that messages will be resent if they do not reach their desired location.

We simulate our protocols with three different network topologies and two different radio ranges. This leads to six independent scenarios, which each of our protocols are tested on. Nodes in the topologies are placed one meter apart in a grid-like manner. Furthermore the topology is built such that nodes with a small node identity are placed closer to the base station. An example of the small topology used in scenario 1, can be seen in figure 5.1. The six scenarios look as follows:

- *Scenario 1*: Small scenario with low density. Topology of size 15x7 meters, for a total of 105 nodes. Radio range 5 meter.
- *Scenario 2*: Medium scenario with low density. Topology of size 25x11 meters, for a total of 275 nodes. Radio range 5 meter.
- *Scenario 3*: Medium scenario with high density. Topology of size 25x11 meters, for a total of 275 nodes. Radio range 10 meter.
- *Scenario 4*: Large scenario with low density. Topology of size 41x11 meters, for a total of 451 nodes. Radio range 5 meter.
- *Scenario 5*: Large scenario with high density. Topology of size 41x11 meters, for a total of 451 nodes. Radio range 10 meter.

The fairly limited radio range was chosen on the basis of early indications from the SensoByg project, concerning a realistic radio range for nodes sending to other nodes from within concrete. In practice this becomes a matter of increased density. As both the protocols under investigation will, in theory, benefit from increased density, this is exactly the property we are looking for in order to compare the gains each of them achieves.

As such, our goal is to have our experiments run on topologies that resemble the bridge scenario in the SensoByg case study, as closely as possible. Thus all topologies are rectangular in shape with the sink sitting in the middle of one of the longer edges of the topology. The nodes are placed one meter apart in a grid-like pattern. The choice of density for the nodes was taken from the SensoByg requirements[4].

Initially we aimed to perform simulations with up to 1000 nodes, but due to the severe memory usage and increased execution time of TOSSIM with larger topologies, this was not possible. See section 4.2.3 for further details. Furthermore, we also performed

91	77	76	75	74	73	72	71	70	69	68	67	66	90	104
92	78	54	53	52	51	50	49	48	47	46	45	65	89	103
93	79	55	35	34	33	32	31	30	29	28	44	64	88	102
94	80	56	36	20	19	18	17	16	15	27	43	63	87	101
95	81	57	37	21	9	8	7	6	14	26	42	62	86	100
96	82	58	38	22	10	2	1	5	13	25	41	61	85	99
97	83	59	39	23	11	3	0	4	12	24	40	60	84	98

Figure 5.1: Example of topology in scenario 1

simulations using the small scenario, with a high node density. It turns out that in this scenario, all nodes are in direct radio contact with the base station. This causes trivial overlay networks in both the GAF and AMCA protocols and makes the scenario a bad benchmark for the actual performance of the protocols. Therefore, the results obtained from the simulations using this scenario are not presented. The remaining five scenarios represents a suitable test setting for our protocols, in the sense that the amount of nodes involved, will result in non-trivial overlay networks. Thus, both GAF and AMCA will be challenged appropriately.

The last thing that needs to be mentioned in regards to setup of the simulations, is the way we handled the issues with TOSSIM that were discussed in section 4.2. As stated, the TOSSIM simulator performs very poorly when simulating large periods of time with larger topologies. Thus, to be able to run our simulations in a realistic timeframe, we had to speed them up. We did this by reducing the granularity of our time scale. Instead of having a second pass every time the simulation had simulated 1000 milliseconds, we have one second pass every 10 simulated milliseconds. The energy model was updated to reflect this fact, and it gives us a speedup of a factor 100. However, this comes at the cost of precision in the simulations, and we had to extend some time intervals to avoid some colliding events, such as several nodes broadcasting messages at the same time. These adaptations are described in the protocol specific sections of this chapter. Our early experiments showed that the speedup does not impact the overall results, and that the precision lost is of minor importance.

5.2 Performance Metrics

In our experiments we are focusing on several different metrics for performance and comparison between the two implementations. One of the most important points of interest

is the ability of each protocol to balance the load of the network across all the nodes, such that no single node will be drained significantly faster than the others. This will ensure that the protocol attempts to utilize all available energy in the network, and will help extend the network lifetime. For a specific simulation we are looking at:

- *Network connectivity loss.* This is the time where the first *live* node in the network loses its connectivity to the base station. A live node is a node that has not yet run out of energy. When this event has occurred, it will become impossible for the routing protocol to route messages from the disconnected node to the base station. Therefore, it does not make sense to consider other performance metrics such as the amount of retries and lost messages after this point in time. Whenever such a metric is evaluated, it is based on data from before the network connectivity loss. Before the loss of connectivity occurs, any amount of nodes may have already run out of energy. This is because of the fact that just because a single node dies, it may still be possible for all the remaining nodes to route their messages to the base station. Therefore it is important to distinguish this performance metric from the next one, which concerns the first node death.
  Because of the reasons above, this metric is also one of the most important ones as it is more or less a direct measure of the routing protocols ability to extend the network lifetime. If compared with the average energy and standard deviation of each node in the network when this even occurs, it is also a measure of the ability of the protocol to utilize all the available energy in the network. In many respects the network can be considered dead at this point in time. In any event it marks the point where the network starts to disfunction.

- *First node death.* Even if the first node running out of energy does not break the connectivity in the network, it still marks the point in time, at which the network will slowly start to degrade, with respect to live nodes. This also marks the first point at which the protocols under investigation might have to find alternative routes to the sink, and thus the ability to perform under less than optimal conditions can be observed in the period from this point on, until the network dies. In itself it is also an early measure of the protocols ability to balance the load of the network.

- *Average lifetime of nodes before connectivity loss.* The *lifetime* of a node, denotes the amount of time that passes before it runs out of energy and dies. This metric is a measure of how long each of the dead nodes managed to stay alive before the loss of connectivity occurs. If this is close to the time where connectivity is lost, it is an indication that the protocol balances the load well.

- *Total unique messages sent and lost.* The percentage of messages lost that the nodes send to the base station, is a direct performance measure of the protocols. As we are running the simulations in a lossy environment, loss is almost unavoidable. However, as the framework contains a retransmission mechanism, and as the protocols are in some cases able to consider alternate routes, this amount should be negligible if the protocols function well.

- *Routing messages sent on average.* The amount of routing specific messages being sent by each node, is an indication of the amount of overhead traffic in the network.

If this average value is high, it may drown the communication channel, making it impossible to route any data messages to the sink. It is especially important that this number does not increase rapidly as the size of the topologies grows. This could hinder the scalability of the protocol. The amount of unique messages lost may also be tied to this metric.

- *Messages routed in total and on average.* This metric is a measure of how many messages in total had to be sent from one node to another in order to route each of the unique messages to the base station. This number depends on the amount of hops each message has to make. This metric will be an indication of the load generated by each unique message that is sent. We will also investigate how many messages each of the nodes route. This will indicate how well the protocol under investigation is able to distribute the load overall. If very few nodes are routing the majority of messages while the others idle, then the load balancing is very poor. We will expect to see a somewhat even distribution of the amount of messages sent, with the nodes closer to the base station routing the most messages as they have to route messages for the majority of the remaining network that is further away. Thus the ability to spread the load evenly among the closest nodes to the base station is especially important if a protocol is to perform well.

- *Retries performed on average.* The amount of lost messages is directly connected to the amount of retries. If there is a substantial amount of retries it is again an indication that there is a lot of traffic blocking the messages being sent.

5.2.1 GAF Specific Experiments and Adaptations

As mentioned in section 4.5.3, GAF relies on a parameter `k`, to determine the discovery times for all the nodes. The higher this value is, the longer all nodes will have to stay active, while deciding on a new leader in a grid cell. However, there is a tradeoff between shortening the discovery time and the amount of retries and lost messages that occur. This is because making the discovery time shorter, means more routing messages will be broadcast in a shorter period of time, possibly blocking the communication channel for any data messages that needs routing. At the same time, a very short discovery time means that both the grid cell leader and the four neighboring cell leaders will broadcast their discovery message almost simultaneously, again, flooding the communication channel. This means that a node that becomes the new leader in a grid cell, might not have received a discovery message from all the neighboring grid cell leaders. This can lead to new leaders trying to route message to nodes that are actually sleeping, which will increase the amount of retries and lost messages drastically.

In order to find the best possible `k`, such that the nodes have to stay active the least amount of time, while still being able to route messages, we performed 3 experiments. The experiments were simulated using a topology of size 15x7 nodes with a radio range of 5 meters. We investigated the performance of GAF with k equal to 500, 250 and 60 seconds. The simulations was stopped when the first node lost connectivity in the network. Thus the performance metrics used to evaluate the experiments were gathered while the

metric / k (seconds)	500	250	60
Connectivity loss (days)	53,53	57,86	57,95
Average lifetime (days)	35,68	38,95	46,58
First node dies (days)	30,97	31,04	19,47
Average retries	15,6	16,8	142,7
Messages lost	49	1	216

Table 5.1: Results of GAF experiments.

protocol was performing under optimal conditions. The results we obtained are summed up in table 5.1.

As the table shows, the time passed before losing connectivity is extended as we lower the value of `k`. The average lifetime of the nodes that die before network connectivity is lost is also increased accordingly. However, it is clear from the tests that having a `k` of 60 seconds causes more inconsistent results. We can observe that there is a dramatic increase in the amount of retries and messages lost. The first node also dies after only 19 days, even though the average lifetime increases. This tells us that the load balancing in the network is being obscured by the situation described earlier, where the short discovery time leads to a flooding of the communication channel by discovery messages.
In the light of these results, we have chosen to use 250 seconds as the default discovery time in all our tests.

5.2.2 AMCA Specific Experiments and Adaptations

There are two things that are specific to the AMCA protocol that need to be investigated: The cycle length and the length of the clusterhead candidate list. The cycle length determines how often the routing protocol updates its routing tree. The more frequently that is done will result in a more correct tree but at the expense of more routing messages.

- Cycle length = 15 hours. This length was chosen because we wanted to see the implications of having the routing protocol update the tree more frequently than the frequency of data messages being sent from the nodes to the base station. This data sending occurs every 24 hours in our application.
- 4 days is about 1/4 of the expected active life time of a node. This means that for every four data messages from a node, it updates the routing tree once.
- Finally, we have the cycle of length of 16 days. This is the expected active life time of a single node.

These three test cases should give us insight into how much the length of the cycle affects the routing protocol and which one of them we are going to use in our simulations. The

outcome of the experiments, can be seen in figure 5.2. Here we see that they perform similar when it comes to first node death. Having a cycle length of 16 days results in that the algorithm can only create the initial tree, then after 16 days we see that the first nodes die since they never manage to move the load onto some other node in the tree. This makes a cycle length of 16 days, unfeasible in our simulations. The other two experiments with cycle lengths of 15 hours and 4 days, have the first node death at similar time and they both lose total network connectivity at the same time. The 4 day cycle length has, however, 29 data messages received on average by the base station compared to 28 data messages in the 15 hour cycle. This information along with the fact that we want to minimize the number of routing messages, results in that we have chosen the 4 day cycle as the cycle length that we use in our simulations.

Another aspect of AMCA that needs to be investigated, is the size of the clusterhead candidate list `P`. The clusterhead candidate list is the list of all possible clusterheads that a node can choose. More nodes in this list, means that more nodes are awake at the same time but a node is more likely to deliver data messages to the base station, since it has more parents to choose from. Having a short clusterhead candidate list is exactly the opposite. Fewer nodes are awake which may increase the lifetime of the network, but nodes have fewer parents to choose from, increasing the chances of messages being lost. Determining a proper size is important to optimize the balance between sleeping and awake nodes.

As can be seen in figure 5.3, then the average energy level of the network is higher with P = 2 compared to P = 4. This is explained by the fact that more nodes are awake with P = 4, hence more energy is used by the network.
If we look at the number of messages delivered to the base station in figure 5.4 and figure 5.5,the difference between the two experiments is not much, at least, by looking at the

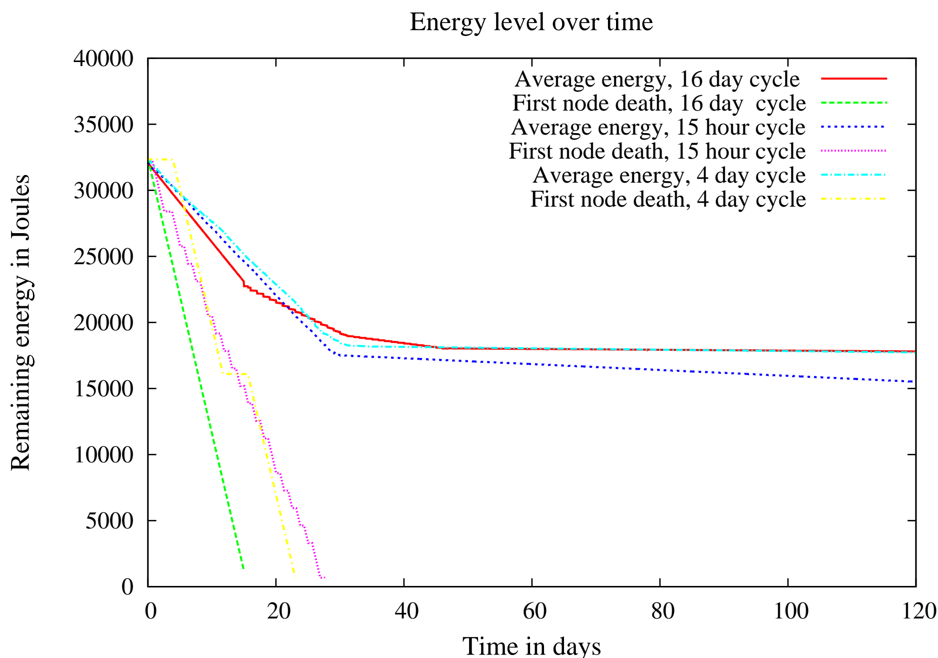


Figure 5.2: Comparison of the energy levels

figures. However, if we look at the average number of messages received by the base station, then we see that P = 2 receives an average of 19 messages while P = 4 receives an average of 17 messages. This makes sense when we see that the average energy of the network is prolonged when using the P = 2. Therefore, we are going to use P = 2 in our simulations.

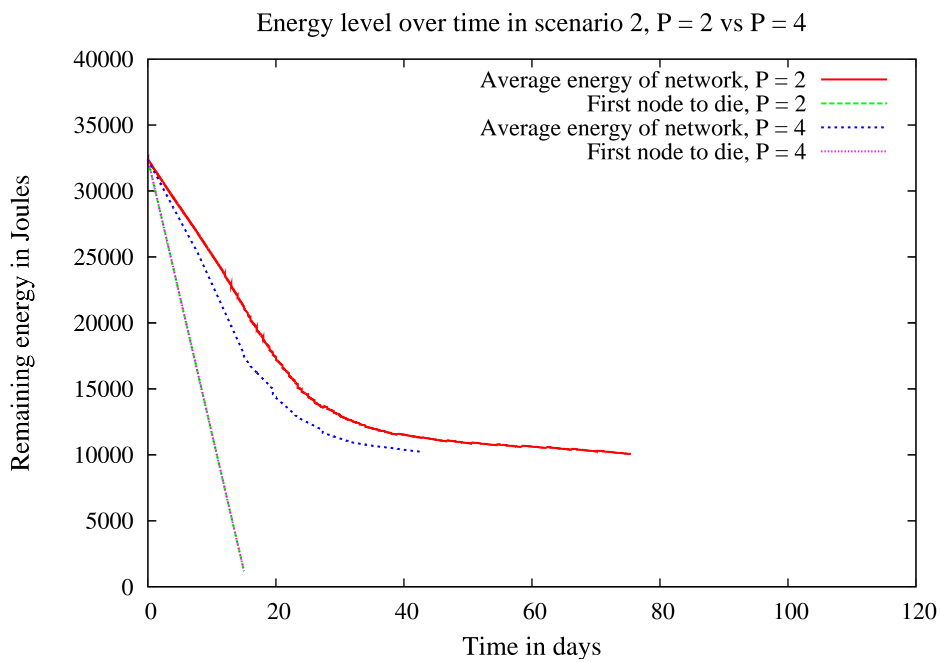


Figure 5.3: Comparison of the energy levels P2 vs P4

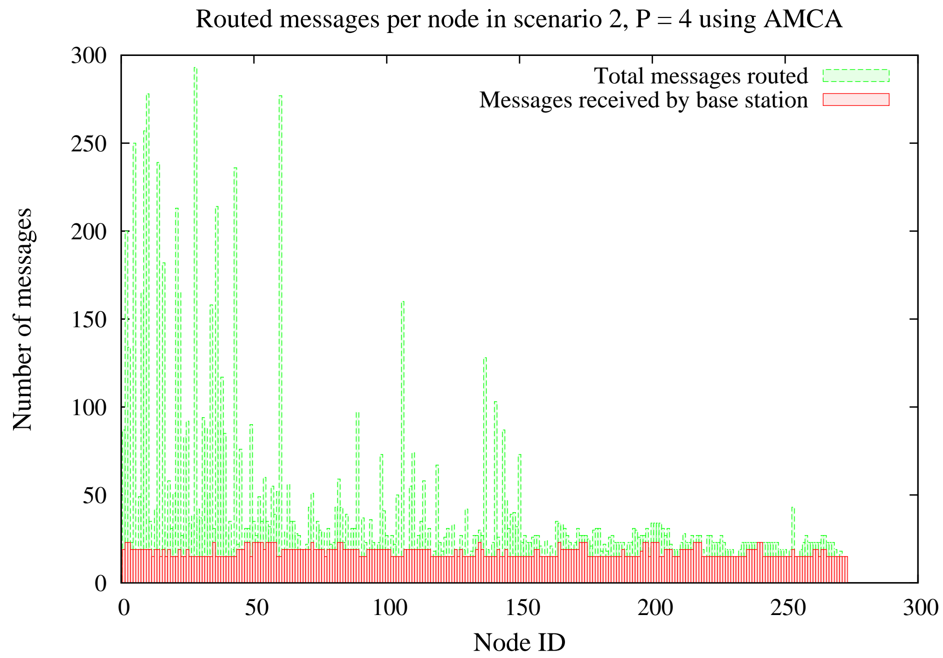


Figure 5.4: Messages sent and received in P4

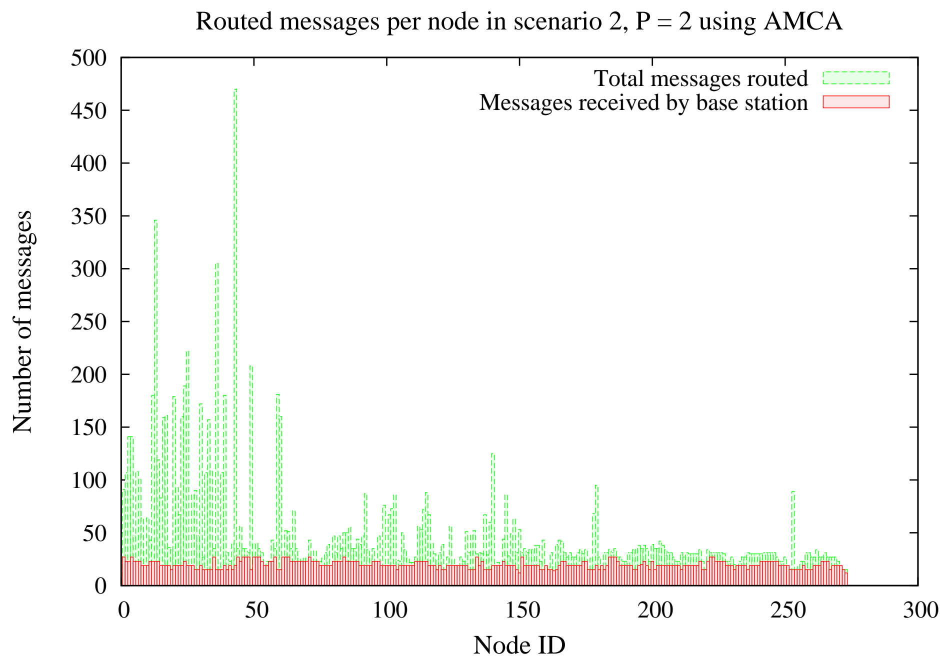


Figure 5.5: Messages sent and received in P2

6

Experimental Results

In this chapter, we describe the results obtained from our simulations. Section 6.1 covers the results from the simulations of the GAF protocol. Section 6.2 covers the results of the AMCA protocol. Finally in section 6.3, we compare the two protocols against each other.

We consider the five simulation scenarios described in section 5.1. As described in section 5.2, we evaluate the protocols with respect to various performance metrics. In order to do this, each test will be accompanied by two figures. The first depicts the average energy of the nodes in the network over time. This is compared to the basic case, where a node is always active, but without sending or receiving any messages, which yields a lifetime of 15,59 days. It is also compared to the case where a node is always sleeping. We expect to see the average energy somewhere in between these two graphs. On the same figure we also plot the energy level of the first node to die in the network, as a comparison to the average. The second figure shows the amount of messages originating from each of the nodes which the base station received as well as the total amount of messages routed by each node. The expected minimum amount of messages, which is also depicted in this figure, is one message for each day that the basic always-active node can stay alive. This will be used to evaluate the protocols ability to balance the load in the network.

6.1 GAF

This section presents the results we have obtained by simulating our implementation of the GAF protocol. Other than the performance metrics described in section 5.2, we also consider a few GAF specific metrics. Specifically, we are interested in certain metrics concerning the grid cells which the GAF overlay network consists of. By analyzing the

lifetimes of the nodes in a specific cell, it is possible to determine how well GAF handles load balancing within a cell. The best case scenario would be if the nodes within each grid cell are able to distribute the load of the network equally among themselves, and thus die around the same time. We therefore consider when the first grid cell in the network dies, in the sense that all of the nodes that reside within the cell have run out of energy.

We also consider how the amount of nodes within a grid cell affects the lifetime of it. We expect that a topology with a high node density will have a positive effect on the lifetime of the network as more nodes will reside within each grid cell.

6.1.1 Scenario 1

Scenario 1 is the small topology size simulated with a low node density. The topology size is 15x7 meters and the radio range is 5 meters. This leads to a GAF overlay network consisting of 7x4 grid cells. As an example of how the network is divided into grid cells, see figure 6.1. The numbers inside each of the grid cells, indicates the cells coordinates in the grid. As is always the case, the base station is located in the grid cell with coordinates `(0,0)`. With this topology, each grid cell consists of 2 to 6 nodes, with the majority consisting of 4. The cells consisting only of two nodes are located furthest away from the base station on the boundaries of the network. Only the cell `(0,0)` where the base station is located has six members. The results we obtained from this test can be seen in table 6.1.

Description	Value
Network connectivity loss	59,77 days
Average lifetime	41,32 days
First node death	31,04 days
First grid cell death	31,04 days (2 nodes in grid cell)
Total unique messages sent / lost	5768 / 7
Routing messages sent total / average	9522 / 90
Messages routed total / average	29763 / 197
Retries total / average	1899 / 18
Average energy at connectivity loss	3505 J
Standard deviation of average	7388 J

Table 6.1: Results for scenario 1 using GAF.

As the table shows, the first node dies after 31 days. This node is located in a grid cell consisting of only two nodes. All nodes within such grid cells die on day 31 of the simulation. This gives them a lifetime roughly twice as long as the basic case where nodes are always in the active state. This is a first indication that the GAF protocol is able to balance the load well inside each of the grid cells.

The network is able to function for almost 60 days before the first node loses connectivity. This is the point where all the grid cells consisting of 4 nodes start to run out of energy and die. In fact, all grid cells with 4 nodes that still have connectivity, die within

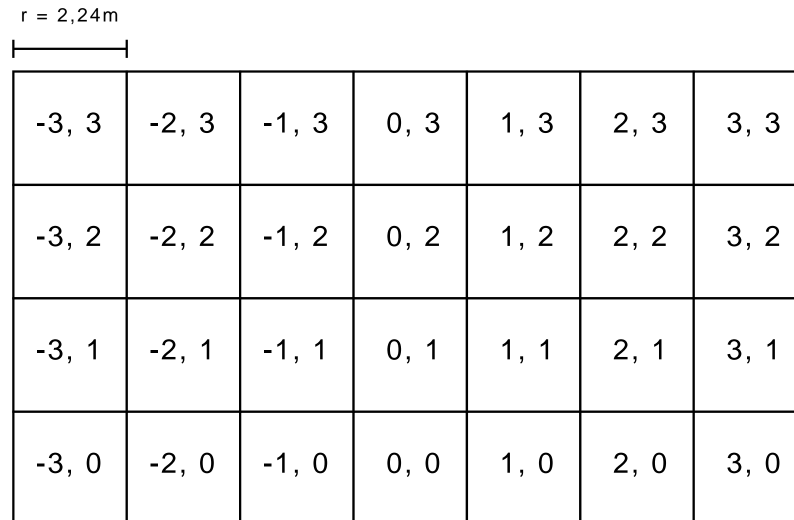


Figure 6.1: GAF grid for the 15x7 node topology.

the interval of 59,77 to 61,70 days. This marks the death of the network as a whole, as only a few nodes directly in contact with the base station are still connected. Again, as each of the four nodes that these cells consist of has enough energy to stay active for 15,59 days, it appears that the lifetime of each grid cell scale almost linear with the amount of nodes in them. This is an indication that the load balancing internally in the cells function very well in GAF.

In order to evaluate the overall load balancing that GAF achieves, we examine the average energy of the network as seen in figure 6.2. At day 59 where connectivity is lost, the average energy is very low at only 3505 joules. This implies that GAF is able to utilize a large amount of the available across all the nodes. However, as stated in table 6.1 the standard deviation is 7388, meaning that most of the nodes at this point have between 0 and 10893 joules. This is a rather large spread, but it still implies that GAF is able to balance the load pretty well across the network, although there may still be room for improvements. Examining the energy of each node closer shows that most nodes have close to zero energy. There are, however, five nodes in the same grid cell as the base station, which always sleeps. These nodes increase the standard deviation considerably as they have a very large amount of energy at the time where connectivity is lost. In fact, if these nodes were to be ignored, the standard deviation would be cut in half. As we will see in the results obtained from the other scenarios, this is the case for all of the GAF simulations.

The reason that the few nodes closest to the base station are always sleeping, is because the base station is always grid leader in their grid cell. It would therefore be possible to further improve the load balancing in the network, by optimizing the way the grid is constructed, such that the base station would be in a grid cell by itself. This can be done by making the nodes in the same cell as the base station part of the closest neighboring grid cells instead. This would also increase the amount of nodes in the cells surrounding the base station which would help increase their lifetime. Improving the grid construction algorithm is left as future work.

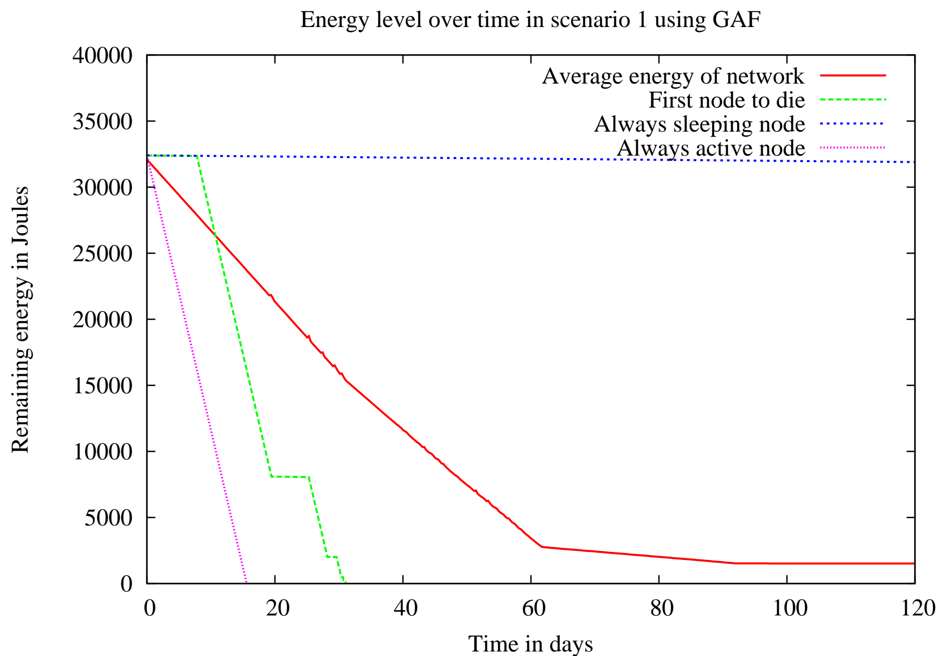


Figure 6.2: Energy over time in scenario 1 using GAF

Figure 6.3 shows how many messages each of the nodes have routed. It also shows how many messages originating from each node, was received at the base station. Nodes closer to the base station will always have to route the most messages as the nodes further away will have to route their messages through these. The topologies used in the scenarios are built such that a nodes identity is an indication of how far it is from the base station, as can be seen in figure 5.1 presented in section 5.1. If we take this into account, the message load is shared fairly even across the nodes in the different ranges from the base station. It is also apparent that all nodes were able to deliver 2-4 times as many messages as the expected minimum of 15 (one for each day the basic always-active node can stay alive).

Looking back at the results in table 6.1, it also appears that the discovery period, where all nodes within a grid cell are awake in order to determine a new leader, does not present any noticeable overhead in the lifetime of the grid cells. The amount of time that each of the nodes have to stay active as they determine the new leader, simply does not seem to have a major effect on the overall lifetime of the grid cells. Similarly, the average of 90 routing specific messages that each node send in the 59 day period, does not present any noticeable overhead. Neither in energy consumption or the ability to route messages from all the nodes to the base station. In fact, with only 7 messages lost on their way to the base station out of the 5768 messages sent in total, the GAF protocol seems to route messages in a reliable manner.

The retransmission mechanism in place in the framework is also functioning as expected, as only 7 messages were lost despite 1899 total retries had to be performed. This includes 108 cases where GAF was asked to find an alternate route for a message that could not be delivered to its next-hop destination.

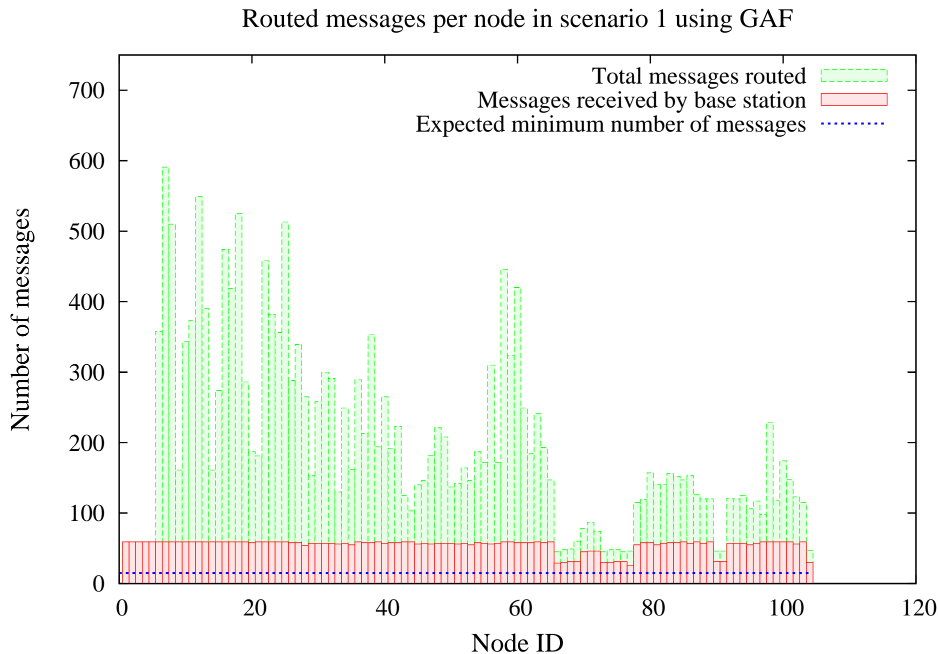


Figure 6.3: Messages routed in scenario 1 using GAF

6.1.2 Scenario 2

This scenario is the medium topology size simulated with a low node density. The topology size is 25x11 meters and the radio range is 5 meters. This leads to 11x5 grid cells in the GAF overlay network. Each grid cell consists of 4 to 8 nodes. In this test, the grid cells with 8 nodes are located in the corners of the topology, with the majority of the other grid cells consisting of 4 nodes. Again, the grid cell `(0,0)` is at the same position as in figure 6.1, but the grid has been extended with one more row and additional columns in each end of the grid. The results obtained from this test can be seen in table 6.2 and figures 6.4 and 6.5.

As in scenario 1, which is the smallest scenario with low density, we observe that the life time of a grid cell seems to scale fairly linear with the amount of nodes within it as the first cell dies after 57 days and it consists of four nodes. Network connectivity is lost on day 57. This is roughly two days earlier than in scenario 1, but as the table shows, the average amount of messages each node has to route has increased from 197 to 326. For the nodes closest to the base station, this means a steep increase in the amount of messages being routed, which is also reflected in figure 6.5. This fact may account for a portion of the two days in difference for the network connectivity loss.

Returning to table 6.2 it is apparent that the average energy of the nodes in the network, was again quite low, when the network lost connectivity. Furthermore, the standard deviation remains almost identical to that of scenario 1. The reason for the spread in the energy is also the same as explained in the results for scenario 1. This indicates that the GAF protocol managed to balance the load reasonably well.

Table 6.2 also shows that the average lifetime of the nodes has increased to 57,38 days, which means that not many nodes have died when connectivity is lost. This is a direct

Description	Value
Network connectivity loss	57,87 days
Average lifetime	57,54 days
First node death	57,38 days
First grid cell death	57,38 days (4 nodes in grid cell)
Total unique messages sent / lost	15532 / 129
Routing messages sent total / average	15188 / 55
Messages routed total / average	89814 / 326
Retries total / average	14203 / 51
Average energy at connectivity loss	8147 J
Standard deviation of average	7894 J

Table 6.2: Results for scenario 2 using GAF.

consequence of the fact that most grid cells consists of four nodes, and that no cells consists of less than four nodes. As a grid cells lifetime is directly dependent on the amount of nodes it consists of, most grid cells therefore die at around 61 days. This point in time can be seen on the graph shown in figure 6.4, where the average energy starts to decrease less rapidly. Again, this is a sign that GAF is able to distribute the load well within each grid cells as it is roughly a factor of four grater than the 15 days lifetime of the basic case.

Even if this marks the death of most of the network, there are still some grid cells with eight members which are still connected. These continue to send messages to the base station until day 91 where the last nodes lose connectivity. This point in time is also reflected in the average energy graph.

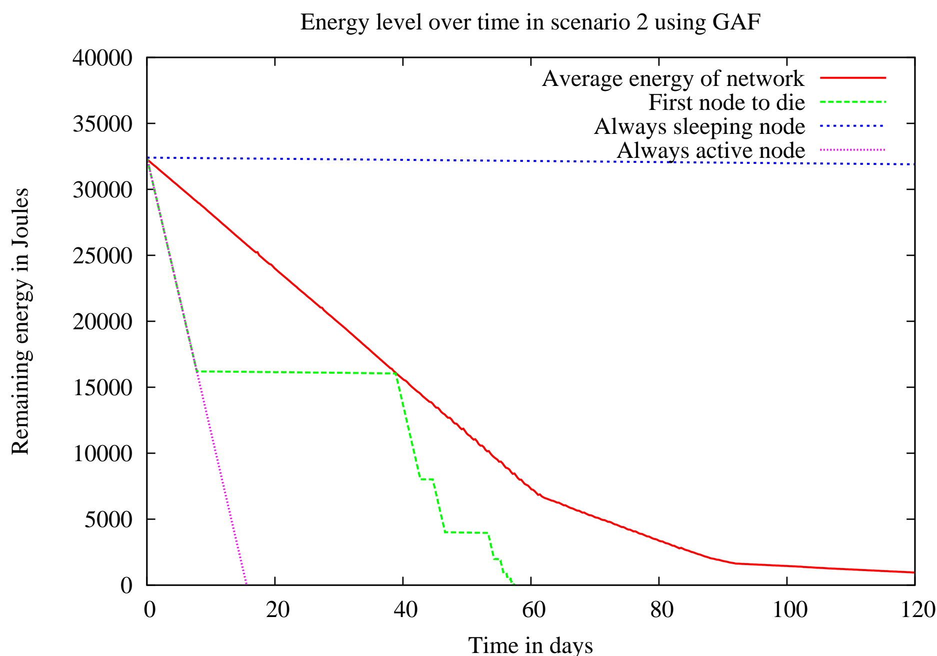


Figure 6.4: Energy over time in scenario 2 using GAF

Before losing connectivity, a total of 129 out of 15532 messages where lost, which is an increase compared to scenario 1. It is, however, still less than one percent of the total messages, which must be considered negligible. The increase may be explained by the increased traffic in the network caused by the additional nodes present in this test. This fact is also reflected in the increased amount of retries that each node had to perform. However, this is to be expected as the greater the traffic present in the network, the greater the chances are that nodes in range of each other will be broadcasting messages simultaneously, thus blocking the radio channel for each other.

In this scenario a total of 15188 routing specific messages was sent which is only an increase of 59% compared to scenario 1. As the topology size has increased by over two and a half times, this is a first indication that GAF is able to scale well with size without flooding the network with routing messages.

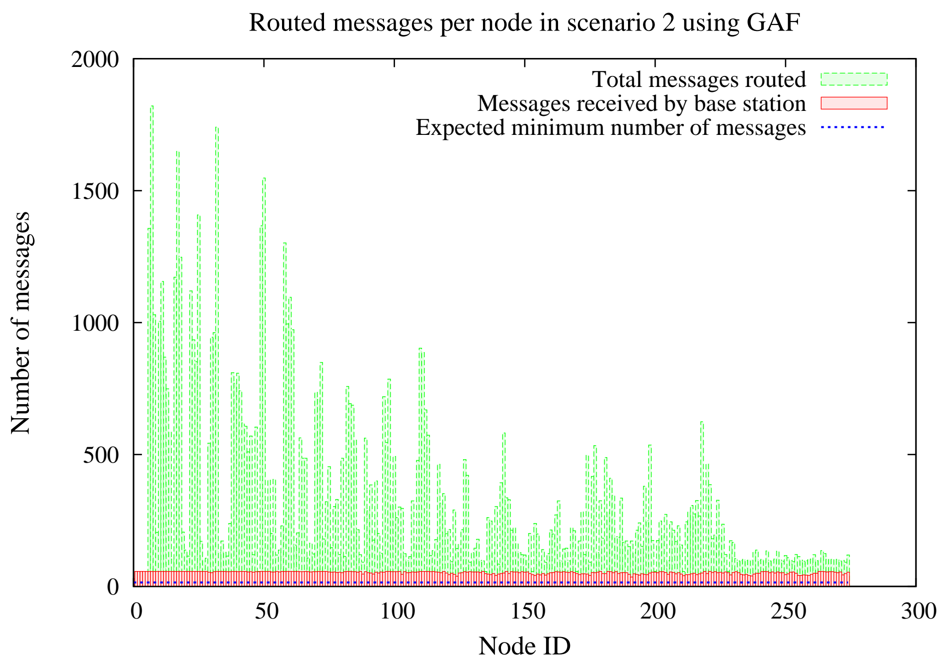


Figure 6.5: Messages routed in scenario 2 using GAF

6.1.3 Scenario 3

Scenario 3 uses the same medium topology size as scenario 2 of 25x11 meters, but with a radio range of 10 meters which yields a high density. This results in an overlay network of 7x3 grid cells, where most grid cells are made up of 12 to 20 nodes. Two grid cells in the corners closest to the base station, only consists of 3 nodes, and the two grid cells in the corners furthest away from the base station is made up of 4 nodes. We expect that these four grid cells die first, but it should not impact the connectivity of the network. It is, however, noteworthy that the distributed grid creation method can result in such uneven grid cell sizes at the edges of the topology. It is unfortunate that there are nodes that will

die considerably before the network loses connectivity, but it is a problem that can be avoided by planning where each node is deployed. This is an option as it is possible to calculate the layout of the grid overlay network, given that each nodes position is known beforehand. Thus it is possible to identify grid cells that will have a substantial less amount of nodes, and configure the deployment appropriately. If the deployment is done in an ad-hoc manner, where the nodes position is not determined until they have actually been deployed, this will of course not be an option. However, in large scale networks the problem of uneven distribution does not have to pose a serious issue, as it is always only at the borders of the topology that the problem arises. Therefore, in very large networks it will only be a fraction of the nodes that will become subject to the issue.

It is in the context of these four special case grid cells that we will now examine the results we obtained from scenario 3. The results can be seen in table 6.3 and the corresponding graphs are shown in figure 6.6 and 6.7.

As argued earlier, we expect GAF to scale well with increased density. This holds

Description	Value
Network connectivity loss	163,66 days
Average lifetime	91,33 days
First node death	46,17 days
First grid cell death	46,41 days (3 nodes in grid cell)
Total unique messages sent / lost	41953 / 310
Routing messages sent total / average	14174 / 52
Messages routed total / average	147388 / 535
Retries total / average	26555 / 97
Average energy at connectivity loss	10974 J
Standard deviation of average	7835 J

Table 6.3: Results for scenario 3 using GAF.

true in this test as the network does not lose connectivity until day 163. At this point all the small grid cells mentioned earlier have died and thus the average lifetime is only 91 days, with the first node dying after 46 days. Again, the nodes within each grid cell die within 1-2 days of each other indicating that the load balancing is functioning well. At the same time, the network becomes disconnected when a grid cell next to the base station dies after 163 days. The grid cell consists of 12 nodes, which means an almost linear increase in the lifetime of a grid cell based on the amount of nodes in it. Without having sent or received any messages, the grid cell could have extended its lifetime to 187 days under optimal conditions. Seeing that grid cells closest to the base station have to route a large amount of the 41953 unique messages sent, the 163 days is an acceptable result.

However the load balancing is not perfect across the entire network. As seen in figure 6.7 there are a few spikes in the messages each node has routed. These spikes occur as several nodes have died before the loss of connectivity occurs. When this happens, some nodes have to find alternate routes for their messages until another node takes over the responsibility of routing from the dead node. This may in turn route a great deal of messages through temporary hotspot grid cells, as it may be the only path to the sink. For the few nodes with the spikes of messages sent, this is exactly what has happened. This

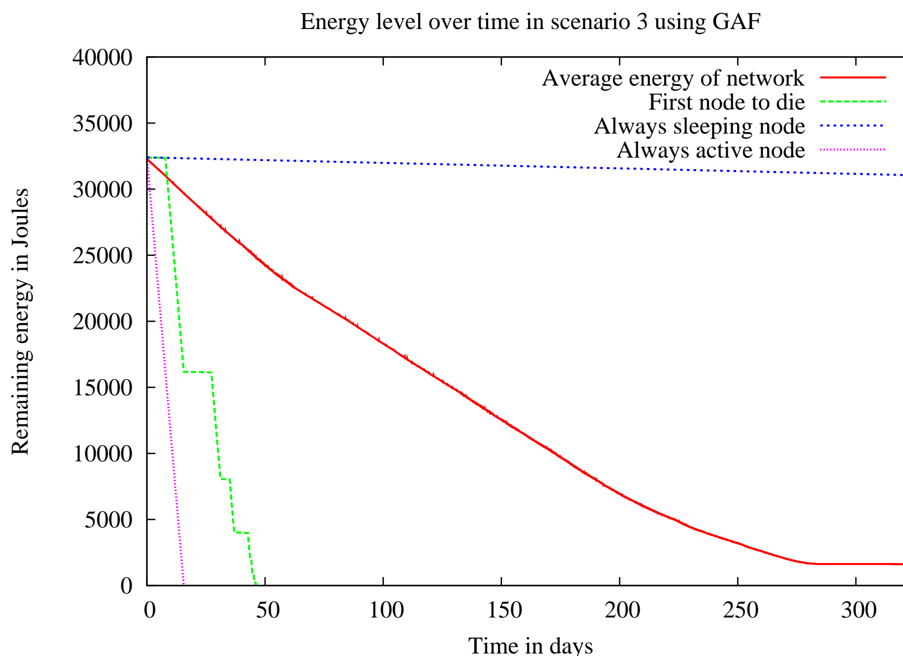


Figure 6.6: Energy over time in scenario 3 using GAF

is also confirmed by the fact that GAF was asked to find an alternate route for a message 5403 times in total. This is to be expected when the network lives for a long period after the first nodes starts running out of energy, as it then becomes less than optimal conditions for the network to route messages in. This can also be seen in the amount of retries each node had to perform compared to scenario 1 and scenario 2. The fact that there is still only 0.74% packet loss shows that the GAF implementation is functioning very well.

6.1.4 Scenario 4

Scenario 4 is a large scenario, using a topology of 41x11 meters for a total of 451 nodes. It is also a low density scenario, with a radio range of 5 meters, which means that the GAF grid consists of 19x5 grid cells. As the density of this scenario is the same as in scenario 1 and 2, which are the other two low density scenarios, the grid cell sizes are also similar to these. Again, this is due to the fact that the size of the grid cells depends only on the radio range.

If the GAF protocol is indeed able to scale to larger topologies we also expect the results from this scenario to be similar to the results from scenario 1 and scenario 2, which uses a small and medium topology respectively. The results obtained from this scenario can be seen in table 6.4 as well as figure 6.8 and 6.9.

After 59 days the network loses connectivity. At this point, the 10 grid cells that only has 2 or 3 nodes are already dead. This leads to a relatively low average lifetime of 38 days. However, compared to scenario 1 and 2, the results look identical in many ways. Scenario 1 also lost connectivity after 59 days and scenario 2 after 57 days. These similar lifetimes

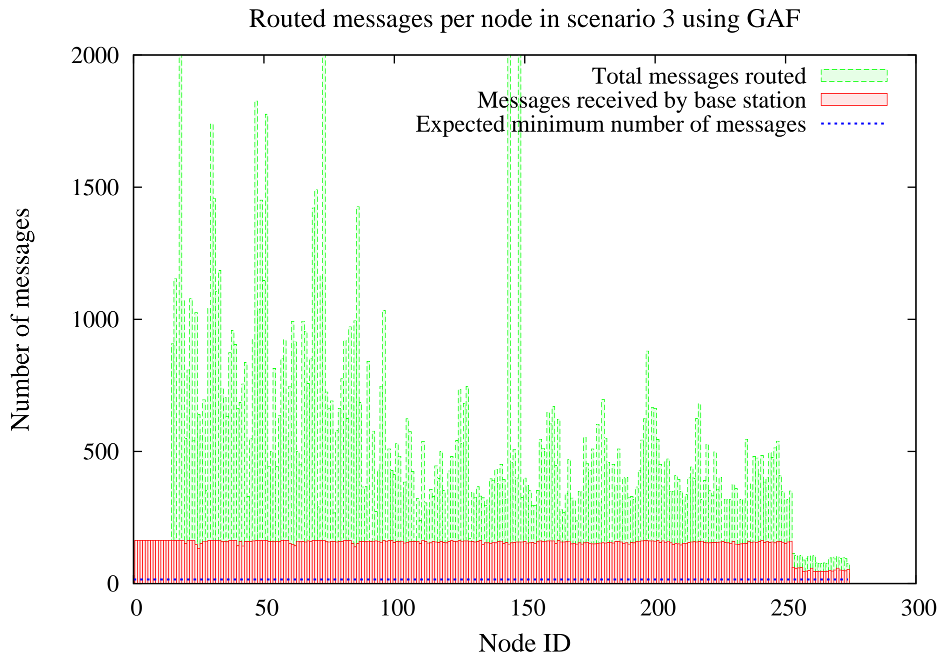


Figure 6.7: Messages routed in scenario 3 using GAF

Description	Value
Network connectivity loss	59,75 days
Average lifetime	38.23 days
First node death	31,04 days
First grid cell death	31,04 days (2 nodes in grid cell)
Total unique messages sent / lost	25760 / 234
Routing messages sent total / average	32725 / 72
Messages routed total / average	195435 / 433
Retries total / average	34036 / 75
Average energy at connectivity loss	7170 J
Standard deviation of average	7826 J

Table 6.4: Results for scenario 4 using GAF.

should be seen in the light of the greatly increased amount of messages that needs to be routed. In this test we see 25760 unique messages being sent, which causes a total of 195435 messages to be routed total. This is much higher than the 9522 unique messages in scenario 1 causing 29763 messages routed in total. The amount is of course dependent on the number of hops each message has to perform before it reaches the base station, which is up to 13 in this case. All in all this tells us that the GAF protocol appears to be mostly dependent on the density of nodes in the network, when it comes to prolonging network lifetime. Thus, the network lifetime benefits the most by increasing radio range of the nodes and by placing the nodes closer together.

For this assumption to hold, it is equally important that scaling the network up in size, does not impact the performance of the GAF routing protocol, in terms of the amount of

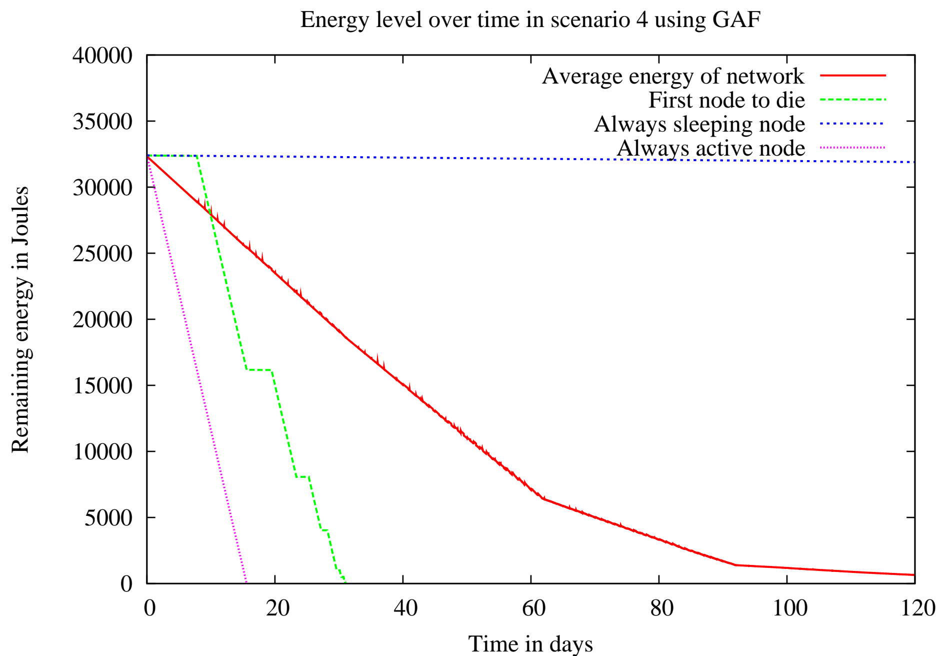


Figure 6.8: Energy over time in scenario 4 using GAF

messages it is able to deliver to the base station. As we can see in table 6.4, 433 messages were lost in this test. This is indeed an increase, compared to the two comparable tests, but it is still less than one percent in total. The increase may be explained by the longer routes that some messages are subject to in the lossy environment that the simulations ran under. But as it is also dependent on which grid cells die first in the different scenarios, it is hard to conclude anything specific about the increase. In any event, the amount of messages lost seems to be fairly negligible, and the increase appears to be minor.

As before, the load balancing appears to be very effective in this test. At the time where the network loses connectivity, the average energy is 7170 joules, roughly 22% of the starting energy, as can be seen in figure 6.8. This implies that GAF was able to utilize a great deal of the available energy across all the nodes, which increases the overall lifetime. Note that the standard deviation is again almost identical to the previous scenarios. The reason for this spread remains the same as described in the results for scenario 1.

The amount of messages routed by each node, also seems fairly evenly distributed across the nodes, depending on how far they are from the base station, as figure 6.9 depicts. As in scenario 3, which is the high density scenario using a medium sized topology, there are a few spikes at some of the nodes that are closer to the base station, but all in all the balancing seems to work well.

6.1.5 Scenario 5

Scenario 5 is the final scenario, and it also uses a large topology of size 41x11, but with a high density as the radio range is 10 meters. This leads to a grid overlay network of size

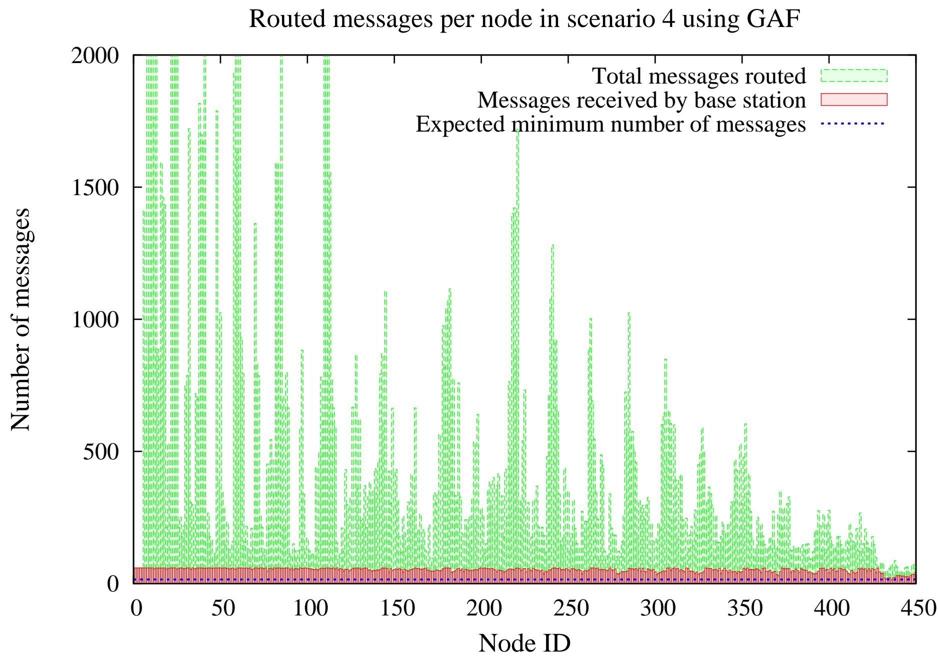


Figure 6.9: Messages routed in scenario 4 using GAF

9x3 grid cells. As this test has the same high density as scenario 3, we expect to obtain results similar to those from that scenario. As the density is the same, the distribution of nodes in each grid cell is also consistent with that of scenario 3. However, as the topology has been extended, there is no grid cells of size 3 in this scenario. The smallest cell is of size 12 and the greatest is of size 20. Therefore we expect the average lifetime to be greater compared to scenario 3. The results can be seen in table 6.5, figure 6.10 and figure 6.11.

As the table shows, the network connectivity is lost as the first grid cell dies after

Description	Value
Network connectivity loss	173,36 days
Average lifetime	173,36 days
First node death	173,35 days
First grid cell death	173,36 days (12 nodes in grid cell)
Total unique messages sent / lost	76018 / 1925
Routing messages sent total / average	18911 / 42
Messages routed total / average	330315 / 732
Retries total / average	47248 / 105
Average energy at connectivity loss	9941 J
Standard deviation of average	7342 J

Table 6.5: Results for scenario 5 using GAF.

173,36 days. This is actually an improvement of the result we got in scenario 3, but this is due to the fact of the more even distribution of nodes in the grid cells. Again, it is noteworthy that the average lifetime of the 12 nodes in the first grid that dies is 14,44 days,

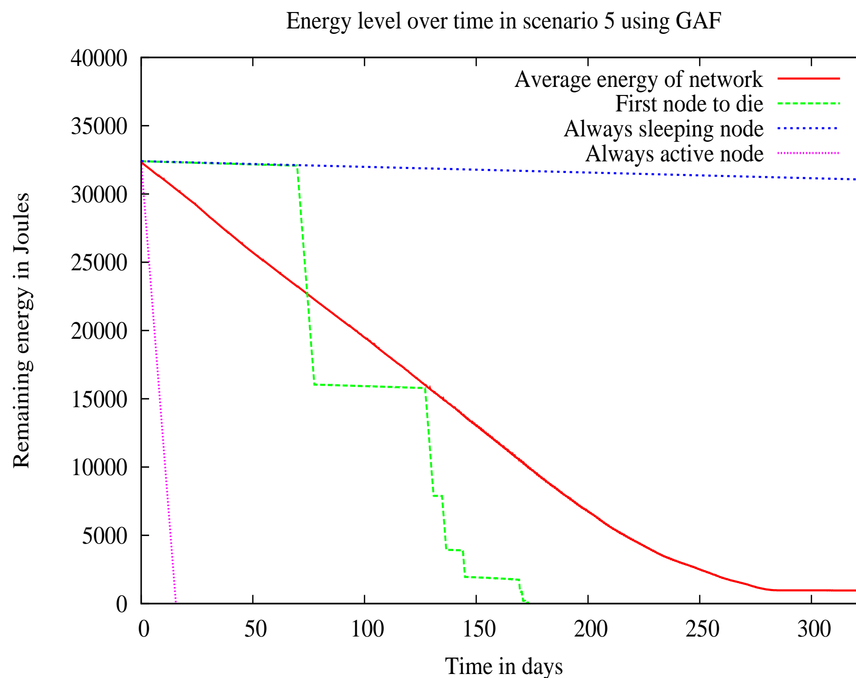


Figure 6.10: Energy over time in scenario 5 using GAF

meaning that the load balancing internally in the grid cells are working as intended. Also, all twelve nodes die almost simultaneously.

It is apparent from all of the tests, that the amount of routing messages being sent in average per node, does not increase dramatically as the topologies grow bigger. In fact it appears that, over time, it is only dependent on the amount of nodes in each grid cell. This is a very nice result, as it is a key element in the scaleability of the GAF protocol. If the average amount of routing messages being sent by each node would increase significantly as the topologies grew larger, they would simply drown the communication channel, making it almost impossible to route any messages to the base station.

There does, however, seem to be some negative impact in the increase of density and topology size. It appears that compared to scenario 3, there is an increase in lost messages. This is the first test to have over 1% lost messages and percentage wise it is an increase of almost 4 times that of scenario 3. Although the 2,5% lost messages can be considered an acceptable amount, it is interesting to explore the cause for this increase.

As the lost messages are spread out pretty evenly across most of the nodes closer to the base station, it does not seem to be caused by any one incident in the simulation. Rather it seems that the increase in the amount of total routed messages, which averages to 732 per node, are overwhelming the nodes closer to the base station, at the specific points in time where the nodes are supposed to send their data message. This might be alleviated by using another displacement algorithm to schedule when each node sends its data message. As it stands now, each node delays the transmission of its message by an amount of seconds, based on its unique node identity. As our topologies are built in such a way that the lowest node identities are closest to the base station and the highest are furthest away, then it can cause uneven *waves* of messages to flow from the nodes to the base station. Especially, as the topologies grow larger, it may become a problem that

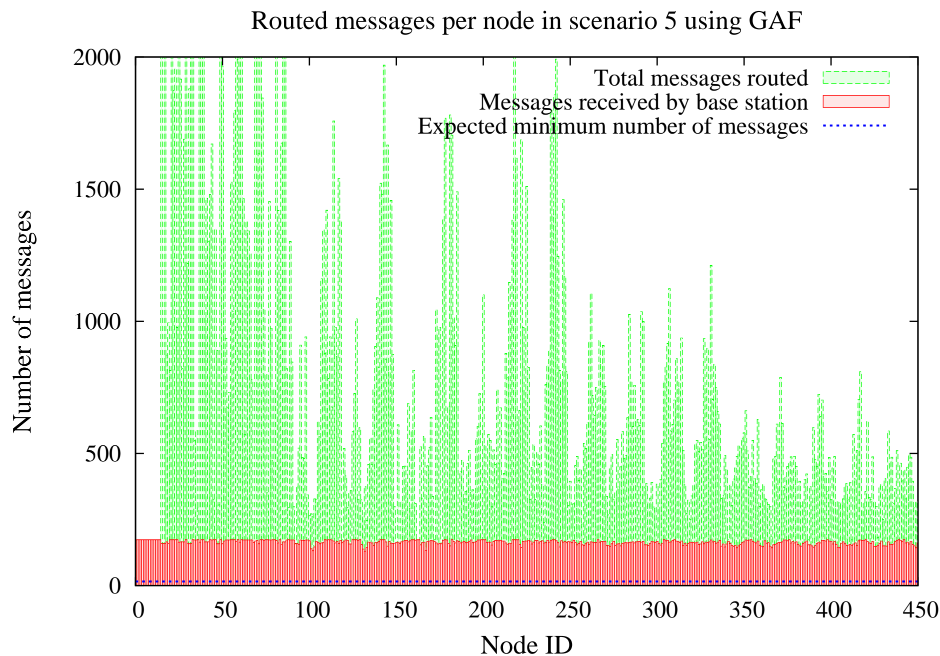


Figure 6.11: Messages routed in scenario 5 using GAF

the nodes closest to the base station sends their messages first and followed by the ones further away. If there is a bottleneck somewhere on the route to the base station it can possibly become flooded with messages in this manner. This is particularly a problem for grid leaders, as members in grid cells will have a tendency to send around the same time. Instead of this time displacement scheme it would be interesting to see if a randomized approach to the problem, would create different results. This could potentially spread the regional times for nodes sending their messages, further apart and could thus be expected to perform better with GAF. This is, however, beyond the scope of this thesis to explore, and is left as future work.

Lastly, it should be mentioned that it is apparent from the graphs in figures 6.10 and 6.11 that the results are very similar to scenario 3. This further confirms the fact that GAF is mostly dependent on the density of the network, and it should scale very well to larger network topologies.

6.1.6 Conclusions on GAF

In figure 6.12 we have gathered all the average energy graphs for GAF. It is noteworthy that the low density scenarios, scenario 2 and scenario 4, have almost identical average energy and standard deviations. Scenario 1, which is the smallest low density scenario, also follows the same tendencies as these two graphs, although at a slightly lower overall average. As such we would expect that a larger topology with the same density would consume more energy, but this does not seem to be the immediate case. Instead, the amount of energy consumed in the network seems to depend on the density of the nodes,

and thus how many nodes are present within each grid cell.

Both scenario 3 and scenario 5 are high density scenarios on medium and large topologies respectively. As is apparent in the figure, the average energy graphs for these two scenarios are close to identical, with the largest topology actually doing a bit better than the smaller one. Again we see that the spread of the energy in the nodes are almost identical to that of the low density scenario. Coupled with the fact that GAF is able to route messages with very little packet loss, and is able to extend the lifetime of the network with up to almost 12 times that of a single node in the active state, we can conclude that GAF scales well with increasing topology sizes. We can also conclude that the protocol is almost entirely dependent on the node density in the network, which of course must be taken into account when considering the applicability of GAF. In terms of being able to scale a network to very large sizes, this can be an advantage as it is possible to makes qualified estimates as to how the energy in the network will be consumed, depending on how dense the topology is.

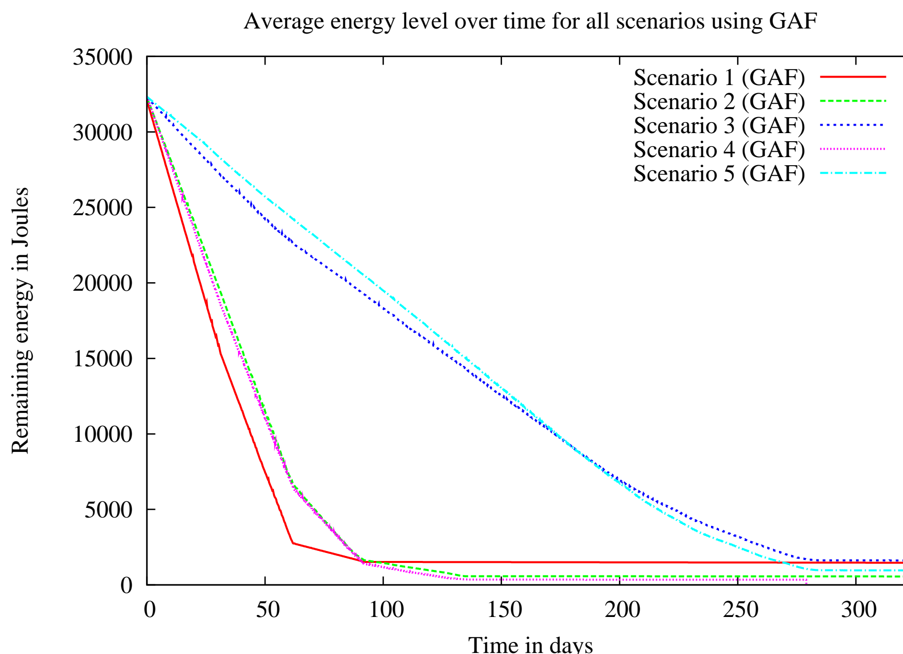


Figure 6.12: Energy over time for all scenarios using GAF

6.2 AMCA

In this section, we describe the results obtained from simulations of the AMCA routing protocol and see how using different sized topologies and densities affects the performance of the protocol. We will also look at how AMCA divides the network into levels in each simulation. Each of these levels contains clusterheads that are parents for nodes in the level below. The clusterheads route data messages from the nodes below to cluster-

heads in the level above. More levels lead to more hops in the network which increases the probability that a message will be lost on its way to the base station.

6.2.1 Scenario 1

Description	Value
Network connectivity loss	27,3 days
Average lifetime	23,4 days
First node death	23,4 days
Total unique messages sent / lost	2776 / 4
Routing messages sent total / average	1458 / 13.9
Messages routed total / average	4401 / 41
Retries total / average	271 / 2.58
Average energy at connectivity loss	19251 J
Standard deviation of average	14616 J

Table 6.6: Results for scenario 1 using AMCA.

This scenario uses the small topology size combined with a low node density. It has 105 nodes spread over 15x7 meters with a radio range of 5 meters. Since the number of levels depends on the topology size and layout and the radio range of the nodes, then this scenario has 3 levels in the network. Level 1 is the largest level since the network is not large enough to create two full levels. How the levels look like can be seen in figure 6.13. The base station, which has ID = 0, is located in the middle of the bottom row. It has the color green to indicate the level it is in. Level 1 has the color yellow and level 2 has the color blue.

In figure 6.14, we see how the energy has been consumed over time in this simula-

91	77	76	75	74	73	72	71	70	69	68	67	66	90	104
92	78	54	53	52	51	50	49	48	47	46	45	65	89	103
93	79	55	35	34	33	32	31	30	29	28	44	64	88	102
94	80	56	36	20	19	18	17	16	15	27	43	63	87	101
95	81	57	37	21	9	8	7	6	14	26	42	62	86	100
96	82	58	38	22	10	2	1	5	13	25	41	61	85	99
97	83	59	39	23	11	3	0	4	12	24	40	60	84	98

Figure 6.13: Example of levels in scenario 1

tion. At 23,4 days, the first node dies. But that does not affect the rest of the network as nodes in lower levels are able to choose another clusterhead to route their data messages. However, at 27,3 days, we have the first network connectivity loss. A node has died at that point and one or more nodes in the lower levels have no longer the option of routing data messages to the base station. Some nodes continue to send messages to the base station, but at day 34 all the nodes in a one-hop range of base station have died. This can be seen in figure 6.14, where the red line becomes horizontal at around 34 days. Overall, the network has only used about 50% of the available energy when the network has lost network connectivity. This is not satisfactory when it comes to trying to extend the life time of the network. When looking at the standard deviation of the average energy in table 6.6, we see that standard deviation is relatively high. This is because we have many nodes that have slept almost the entire time and some nodes that have no energy. This shows that the protocol has not managed to balance the load of the network in this scenario.

In figure 6.15, we can see the amount of messages that each node had to route. It is apparent that nodes closer to the base station route more messages than nodes in the lower levels. The reason that nodes 49, 59 and 60 are routing more than their neighborhood nodes, is due to the layout of the topology. Since they can reach the base station in one hop, they are in level 1. The nodes around them are in level 2. We see that the messages routed is distributed fairly among the nodes in level 1 in the sense that no one node in level 1 is routing all the messages for nodes in level 2. The nodes in level 2 do not route any messages for any other node since level 2 is the lowest level in this topology.

Based on the results in table 6.6, we see that the implementation of the routing protocol is functioning well since the number of lost messages is very low compared to the number of messages sent. The number of retries performed is also low given that we are simulating in a lossy environment. This shows that the AMCA routing messages does not present any significant overhead to the communication channel, which is vital to the ability to route data messages. This is also reflected in the fact that each node only sent 13,9 routing messages on average before the loss of connectivity.

6.2.2 Scenario 2

This scenario is a medium sized topology, with 275 nodes spread out over 25x11 meters. The radio range is 5 meters. This makes it a medium size, low density scenario. The number of levels in this simulation is 4, with levels 1 and 2 being the largest ones.

In table 6.7, we have the results for scenario 2. Here we notice that the time of first node death is at 15,6 days. That means that at least one node was awake the entire time since the expected life time of an always active node is around 15 days. The reason for this can be found by looking at the levels created by the protocol. To illustrate the problem, we look at an example topology, found in figure 6.16. Here we have an example of how AMCA have divided the network into levels. Each level has a different color with green being level 0 and red being level 1 and so on. In this particular case, the radio range has been set to 2 meters but the same principle will apply to all radio ranges. Node 39 (lower left corner) can only reach one node in the level above, namely node 11. This means that node 11 is going to be active all the time, as it will always have to act as a clusterhead for

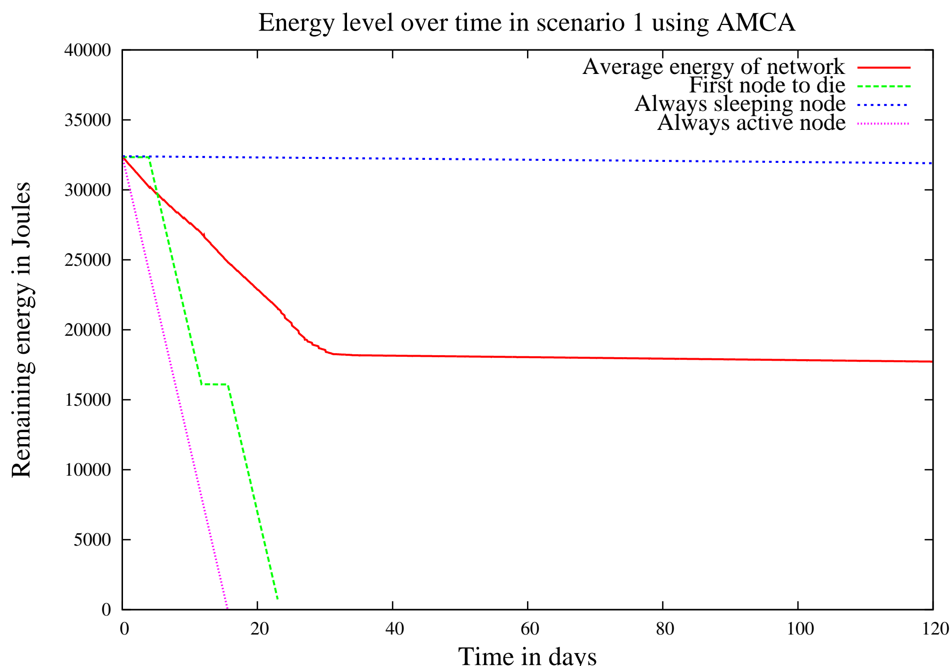


Figure 6.14: Energy over time in scenario 1 using AMCA

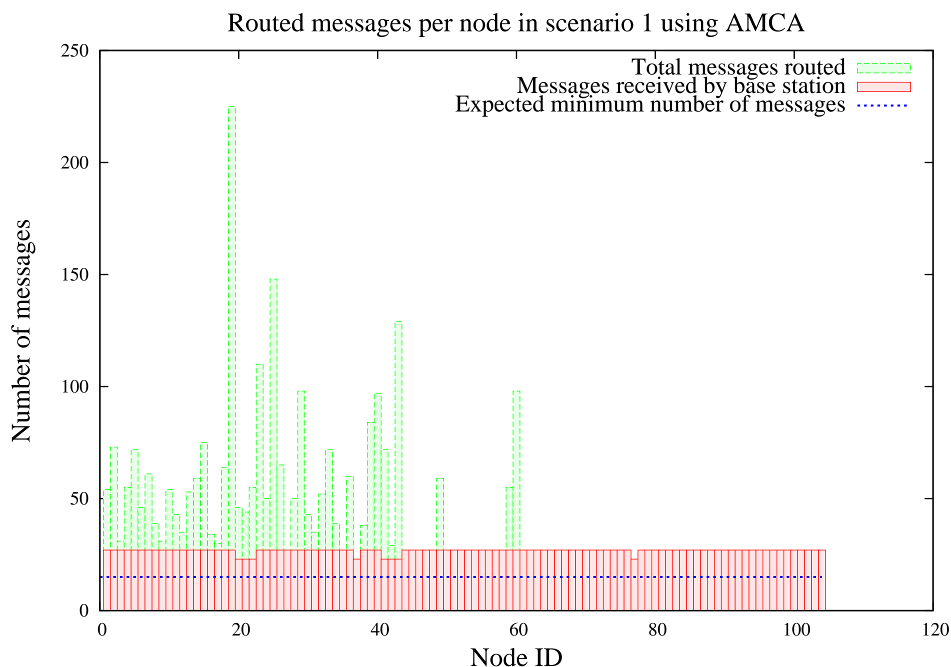


Figure 6.15: Messages sent in scenario 1 using AMCA

node 39. Hence node 11 dies after 15,6 days. This also means that we start losing network connectivity at this point in time, as node 39 now has no way of routing its messages to the base station.

In figure 6.17, we can see the energy consumption of the simulation. The line for the first dead node follows exactly the line for an active node. The average energy of the

Description	Value
Network connectivity loss	15,6 days
Average lifetime	15,6 days
First node death	15,6 days
Total unique messages sent / lost	4104 / 9
Routing messages sent total / average	2200 / 8
Messages routed total / average	9290 / 33
Retries total / average	579 / 2,1
Average energy at connectivity loss	20755 J
Standard deviation of average	10000 J

Table 6.7: Results for scenario 2 using AMCA.

network decreases at a steady rate until the first nodes die at around 15,6 days. At around 35 days, all nodes in the one-hop range of the base station have died, rendering the entire network unreachable from that point on. At around day 95, the simulation was stopped. At the point where the network loses connectivity, the average energy of all the nodes are still very high,as can be seen in table 6.7. Combined with the fact that the standard deviation shows that the energy of individual nodes is spread out pretty far, shows that the protocol was not able to balance the load of the network very well.

In figure 6.18, we see how the messages have been distributed amongst the nodes. All nodes only managed to send the minimum number of data messages since the network lost connectivity at around 15 days. The distribution of the message load is relatively fair amongst the nodes. This is because most of the nodes in the lower levels could choose between different clusterheads and therefore spread the load. As we can see in the figure 6.18, there are a few spikes in it. This is because the nodes with the spikes, were the only few available clusterheads for some nodes in level 2. This means that they had to route many messages for these nodes.

In table 6.7, we see that the number of messages lost is low compared to the total number of messages sent. Also the number of retries is negligible for that period. Again, this shows that the AMCA implementation was able to function well with respect to delivering messages to the base station. When we compare scenario 2 to scenario 1 (which is also a low density scenario, but with a small topology), we see that the number of routing messages sent, does not seem to increase rapidly as the topology size increase. This is due to the fact that a node will always send only two routing messages per cycle in the AMCA protocol. This means that AMCA scales well in the sense that increasing the number of nodes in the network does not lead to an exponential growth in the amount of routing messages sent.

6.2.3 Scenario 3

This scenario has the medium size topology, 25x11 meters for a total of 275 nodes, each with 10 meters in radio range. This makes it the medium sized, high density scenario. The

55	45	35	34	33	32	31	30	29	28	44	54
56	46	36	20	19	18	17	16	15	27	43	53
57	47	37	21	9	8	7	6	14	26	42	52
58	48	38	22	10	2	1	5	13	25	41	51
59	49	39	23	11	3	0	4	12	24	40	50

Figure 6.16: Example of level division in AMCA

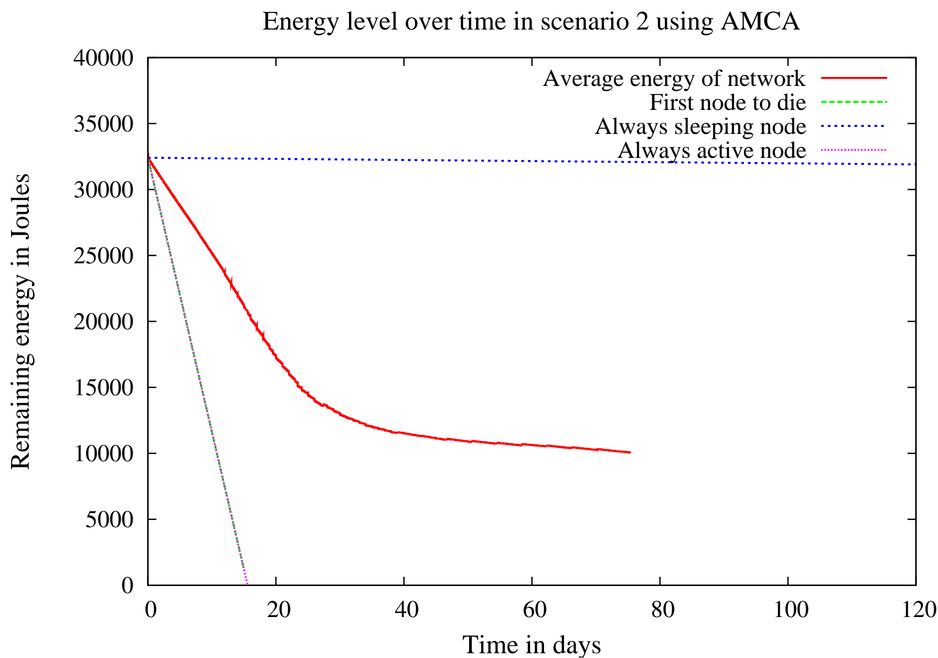


Figure 6.17: Energy over time in scenario 2 using AMCA

AMCA protocol has divided the topology into two levels with level 1 being the largest. Given our simulation timeframe and the performance issues of TOSSIM, explained in section 4.2.3, this simulation was not able to complete. We are still able to present the preliminary results that we have obtained in the simulation before it was stopped. However, this means that there is no time for first node death and no network connectivity loss since both these events had not happened in the scenario when the simulation was stopped.

In table 6.8, we have the results for scenario 3. As discussed above, there is no first node death nor connectivity loss when the simulation was stopped at day 43. That means that the protocol has managed to make all nodes stay alive for 43 days. In that period, the nodes have sent a total of 11725 messages and only 50 messages were lost. That means that 0,42% of the messages were lost which can again be considered negligible. It is also

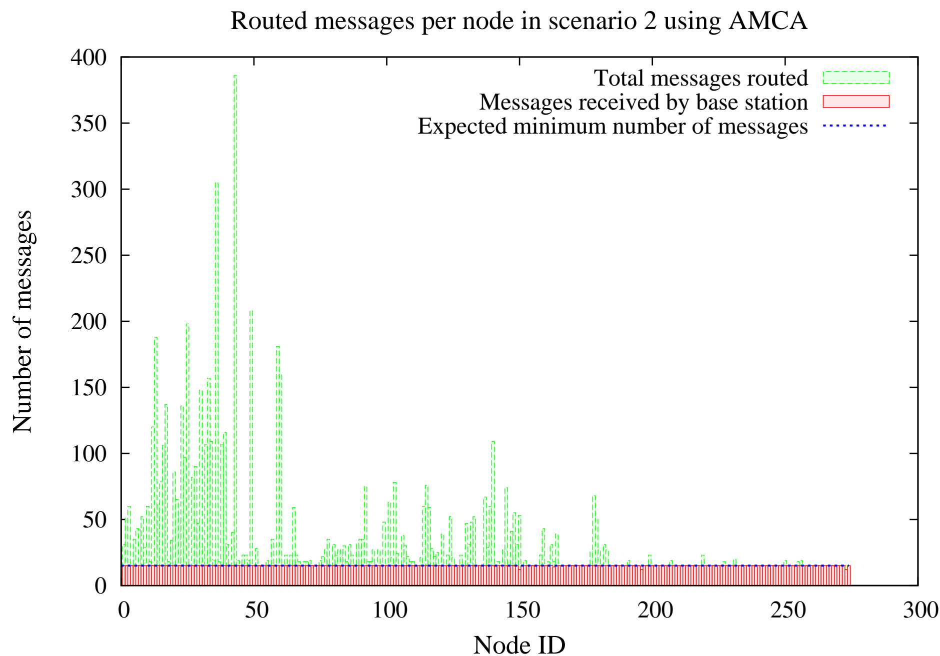


Figure 6.18: Messages sent in scenario 2 using AMCA

apparent that the amount of retries that each node had to perform on average, is almost identical over time to that of the smaller scenarios 1 and 2, which have a lower density. This again implies that the routing messages does not impact the protocols ability to route messages to the base station, even when the density and topology size is increased.

Description	Value
Simulation stopped	43 days
Average lifetime	
First node death	-
Total unique messages sent / lost	11725 / 50
Routing messages sent total / average	6050 / 22
Messages routed total / average	16274 / 59
Retries total / average	1379 / 5
Average energy at simulation stop	25722 J
Standard deviation of average	6439 J

Table 6.8: Results for scenario 3 using AMCA

In figure 6.21, we see how the energy has been used over time. As discussed earlier then there is no first node death in this graph, but we see that the average energy is very high over time and decreases at a steady rate. When the simulation was stopped at day 43 then the average energy of the nodes was 25722 J. The standard deviation of the average energy is 6439. This means that most of the nodes are somewhat close to the average energy and that the protocol manages to balance the load of the network better than in the previous low density scenarios. In fact, at the point where this simulation was stopped, it

had already surpassed the lifetime of scenario 1 and 2. This is an indication that AMCA seems to benefit from higher density scenarios. It is also noteworthy that, as in scenario 1, this scenario does not suffer from the same problem with the level division created by AMCA, as scenario 2 does.

In figure 6.22, we have the message distribution of the network. As the simulation had not finished, it is hard to conclude on anything from this graph, as some nodes may not have had the chance to become clusterheads yet. This means that the distribution of the messages when the simulation would have lost connectivity, might differ greatly from the tendencies seen in this figure.

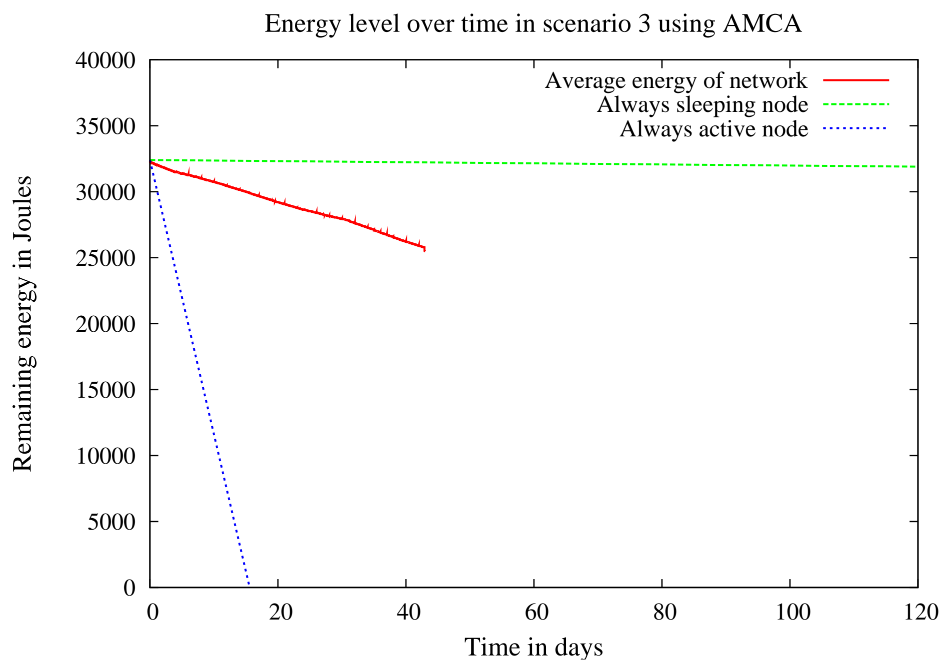


Figure 6.19: Energy over time in scenario 3 using AMCA

6.2.4 Scenario 4

Scenario 4 is the large topology coupled with a low density. The topology is 41x11 meters in size for a total of 451 nodes. The radio range is 5 meters. AMCA divides this network into 6 levels, with level 1 and 2 being the largest ones.

In table 6.9, we see the results of the simulation. The first node death occurs at 15,6 days, which means that some nodes never got the chance to go to sleep. The reason for this is similar to that described in section 6.2.2. We also start losing network connectivity at that point, since there are nodes that are dependent on those nodes and have no choice of other clusterheads. There are no lost messages despite the fact that 6750 messages were sent and we had 825 retries. This shows that if we increase the amount of nodes in the network then that does not automatically lead to an increase in lost messages. Also,

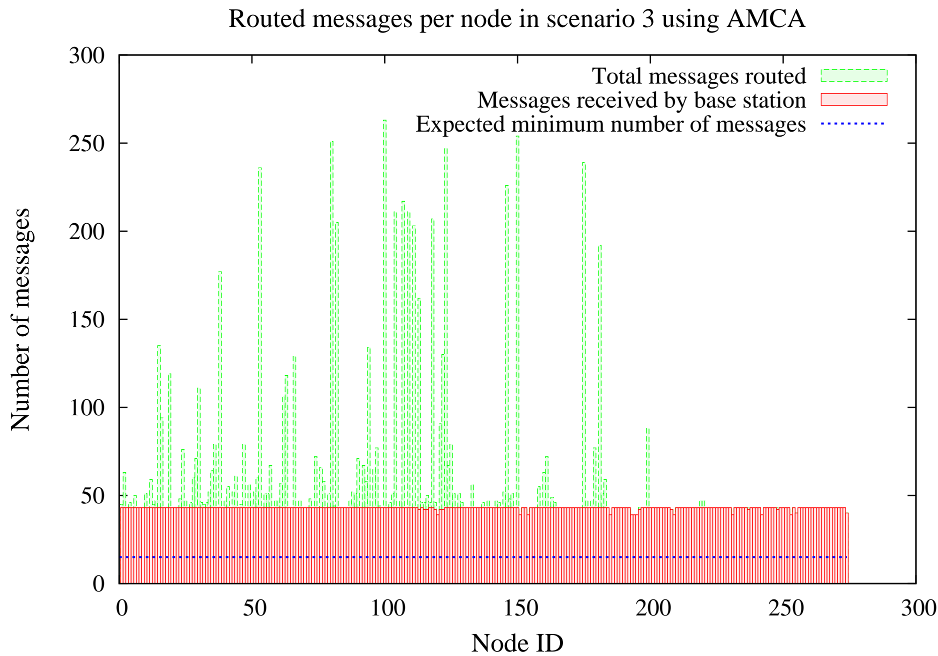


Figure 6.20: Messages sent in scenario 3 using AMCA

as explained earlier, increasing the topology size does not introduce a rapid growth in the amount of routing messages being sent. These two facts are both important for the scalability of the protocol.

Description	Value
Network connectivity loss	15,6 days
Average lifetime	15,6 days
First node death	15,6 days
Total unique messages sent / lost	6750 / 0
Routing messages sent total / average	2200 / 8
Messages routed total / average	20383 / 45
Retries total / average	825 / 1,8
Average energy at connectivity loss	18207 J
Standard deviation of average	9961 J

Table 6.9: Results for scenario 4 using AMCA.

In figure 6.22, we see how the messages were distributed among all the nodes. No node sent more then the minimum number of messages. That is because we started losing network connectivity at that point. There are a few spikes in the number of routed messages and the nodes that have those spikes are usually in the outer edges of level 1. That is the level that will route the most messages since it is closest to the base station. The reason why it is usually the nodes on the outer edges of that level is because they are reachable by more nodes in level 2 than other nodes in level 1. This also means that they usually die faster because these nodes have to stay awake most of the time. Besides these

spikes, there is a fair distribution of the load amongst the nodes, meaning that the routing protocol manages to route the traffic through different nodes at different times. This could lead to longer life times if it were not for the fact that some nodes can only choose so few clusterheads.

In figure 6.21, we see how AMCA has done during this simulation. Besides the fact of first node death at 15.6 days, it manages to use the energy available in the network, better compared to scenario 1 and scenario 2. These two scenarios have low and medium size, respectively and they both have low density. However, as is apparent from the high average energy and the large standard deviation at the time where connectivity is lost, AMCA does not seem to balance the load very well.

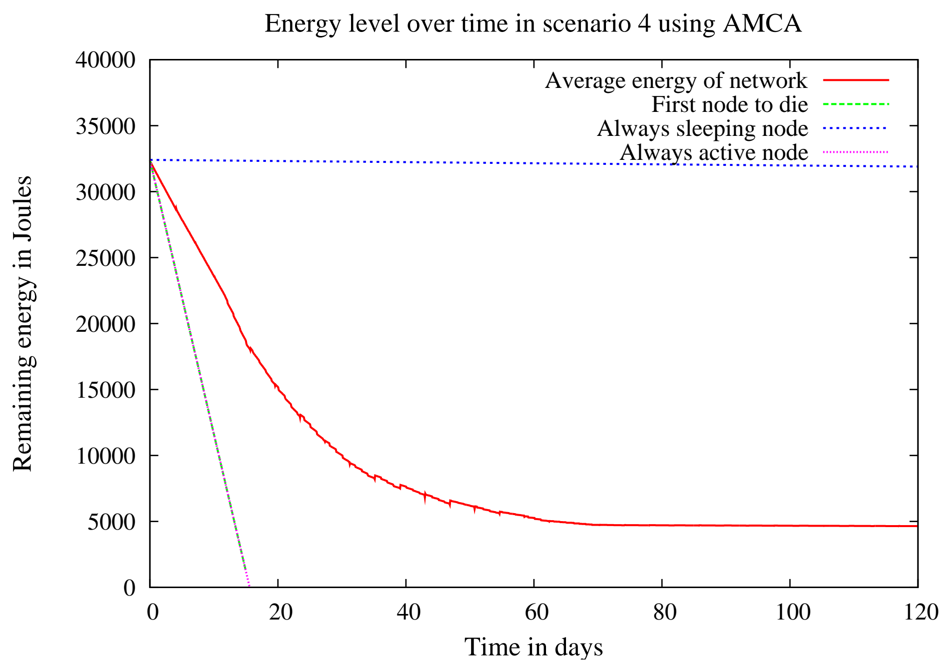


Figure 6.21: Energy over time in scenario 4 using AMCA

6.2.5 Scenario 5

Scenario 5 is the final simulation. It has the same topology as scenario 4, 41x11 meters in size for a total of 451 nodes. However, the radio range is 10 meters, making it a large scenario with high density. AMCA divides the network into 4 levels, with level 1 being the largest one. The reason why there are fewer levels in this simulation compared to scenario 4 (which is large in size with low density), is because of the longer radio range. This also means that each level will have more nodes.

In table 6.10, we have the results for this simulation. Again, we see that the first node dies at 15,6 days. The reason for that is the same as in the previous simulations where this has occurred. This also means that the network loses connectivity at this point. Again, this shows that the scalability of AMCA is very limited in the scenarios we are simulating.

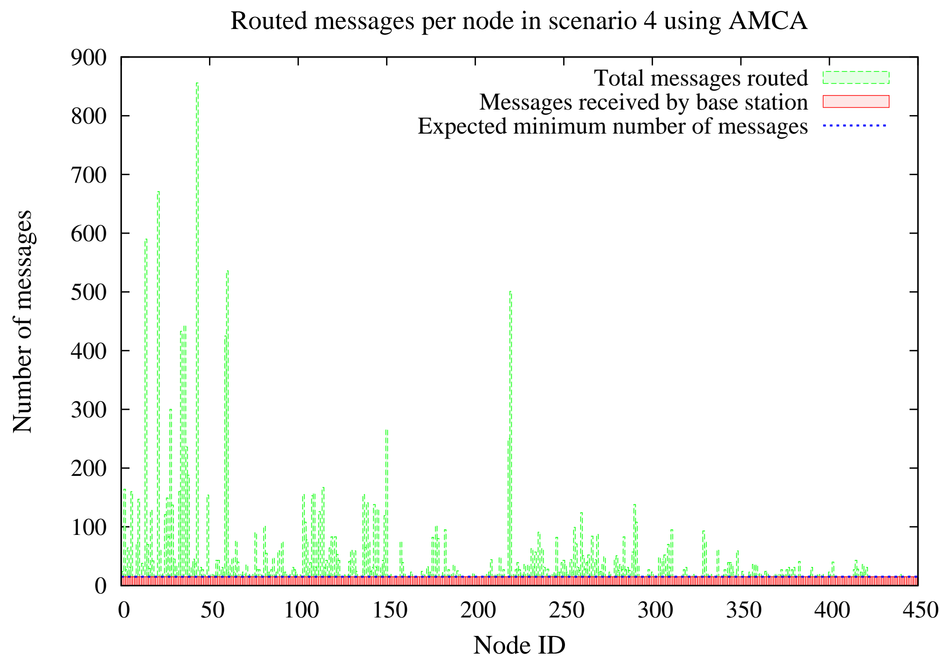


Figure 6.22: Messages sent in scenario 4 using AMCA

There are a total of 6720 unique messages sent in this simulation and only 45 messages lost. When compared to all the previous scenarios it is clear that AMCA is able to deliver its messages to the base station in a reliable manner. Even if both the topology size and node density increases.

Description	Value
Network connectivity loss	15,6 days
Average lifetime	15,6 days
First node death	15,6 days
Total unique messages sent / lost	6720 / 45
Routing messages sent total / average	2200 / 8
Messages routed total / average	11501 / 25
Retries total / average	1897/ 4,2
Average energy at connectivity loss	27849 J
Standard deviation of average	10773 J

Table 6.10: Results for scenario 5 using AMCA.

In figure 6.23, we see how the load has been distributed among the nodes. It is clear that most nodes only sent their own messages, and did not route any messages for other nodes. This is because a large number of the nodes are within 1-hop range of the base station. There are, however, a few nodes that route many messages. These nodes died at 15,6 days and quite some number of nodes in level 2 were using those nodes in level 1 as their intermediate nodes. Again, this points to the problem described earlier, where some nodes will always be active since they will always have to be clusterheads for some

nodes in the level below. This problem is one of the major issues that prevents AMCA from obtaining any form of scalability in our scenarios. When looking at figure 6.24, we see that even though the network lost partial connectivity early in the simulation, it still consumes a steady amount of energy until around day 280. This is because only parts of the network has become disconnected, and some nodes are still able to route messages through intermediate nodes to the base station. That shows that the protocol is able to prolong at least a partial functionality of the network for a greater period of time then in the low density scenarios. However as the standard deviation seen in table 6.10 the spread of the nodes energy is very large, indicating that the load balancing is still very poor.

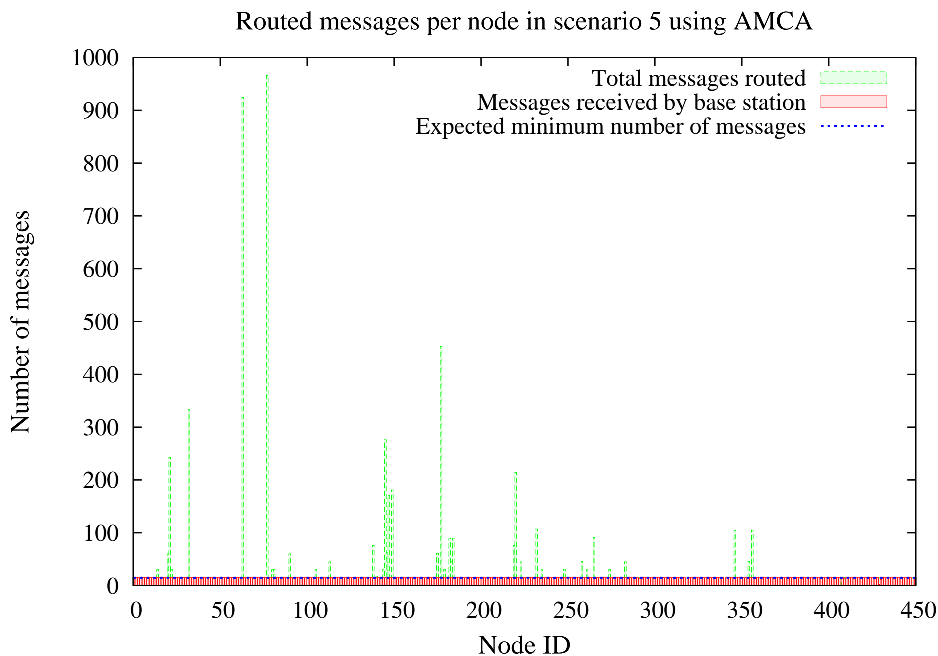


Figure 6.23: Messages sent in scenario 5 using AMCA

6.2.6 Conclusion on AMCA

When looking over the results obtained from simulating the five scenarios with AMCA, it is apparent that AMCA benefits from greater node density. As we can see in figure 6.25, scenario 3 and 5 (which are the high density scenarios with a medium and large topology respectively) deplete the average energy of the network at a slower pace. Even though, scenario 3 did not manage to complete, it had still not lost connectivity after 43 days. Scenario 5 lost connectivity after 15 days, but it was still able to maintain partial connectivity in the network. If it had not been for the unfortunate issue of some nodes always having to be active in scenario 5, it appears that it could have benefited from the increased density as well. In fact, only scenario 3 and scenario 1 (which is the small topology with a low density) were not affected by this issue. Thus, they are also the two scenarios which show the most promising results.

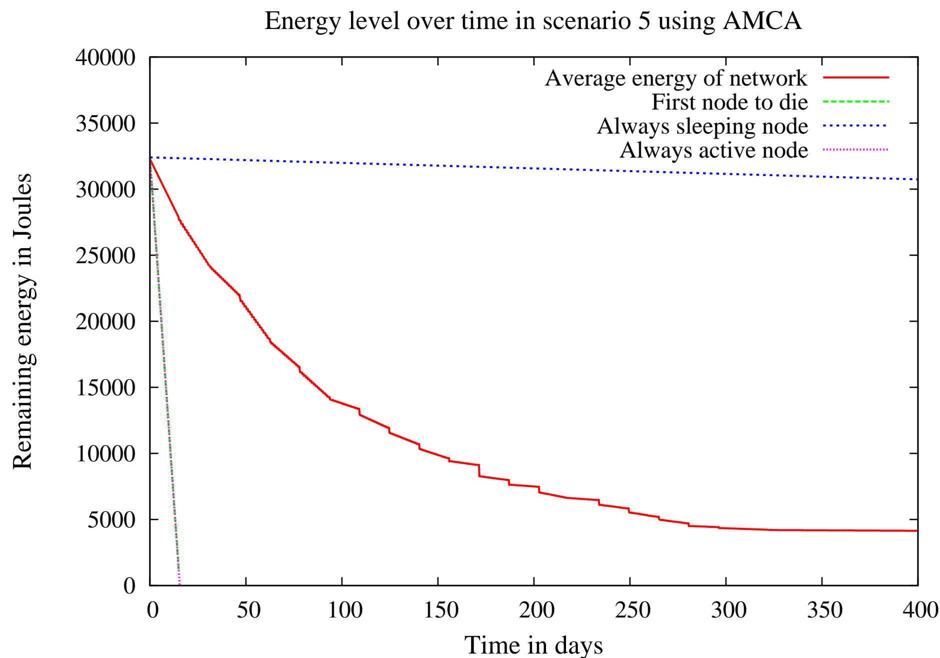


Figure 6.24: Energy over time in scenario 5 using AMCA

However, even though scenario 1 did not suffer from the problem, and was therefore able to utilize a greater amount of the average energy in the network before losing connectivity, the 27,3 days it managed to extend the lifetime to is hardly impressive in the context of SHM. As the remaining scenarios were somewhat crippled by the same issue, it goes to show that some tweaks would have to be made to AMCA before it can become widely and easily applicable to a scenario such as the one we have presented from the Sensobyg project.

However, could these issues of load distribution be overcome, AMCA does show promise in the fact that it is robust in delivering messages to the base station and that it presents very little overhead in the form of routing messages. Although as we have seen in our results, it is unlikely that a tree based protocol such as AMCA could truly be made useful in a setting such as ours.

6.3 Comparison of the Protocols

In figure 6.26 and 6.27, the average energy graphs from our low density and high density scenarios are depicted. To ease comparison of the protocols, each of the energy graphs has been cut off at the point where the connectivity in the network was lost in the simulated scenario. Table 6.11 and 6.12 also sums up the main results for the low and high density scenarios respectively. In these tables *S1-S5* stands for scenario 1 to scenario 5. For more detailed results we refer to the individual scenario result sections.

In comparison, it is clear that GAF is able to utilize the available energy in the network better than the AMCA protocol. In all tests with a low density of nodes, AMCA depletes

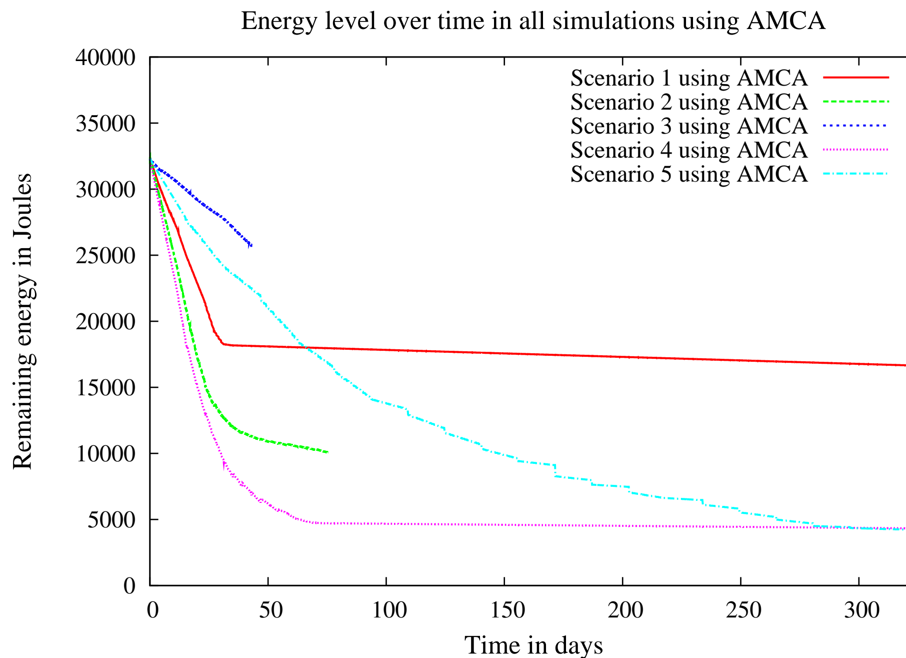


Figure 6.25: Energy over time for all tests using AMCA

its energy at a more rapid pace then GAF. This tendency can be seen in the graphs in figure 6.26 and the results in table 6.11. At the same time, connectivity is lost very early in AMCA, which means that the average energy of the network is still high. GAF is better capable of maintaining connectivity in the network until the average energy becomes low. At the same time, the standard deviations from the average energy at connectivity loss, shows that AMCA has a larger spread in the amount of energy each node has. Combined, this indicates that GAF is better able to balance the load across the network.

The same is true for the results from scenario 3 and 5, where the density is high, as seen in figure 6.27 and table 6.12. It is apparent that both protocols benefit from the increased density, but again the GAF protocol was able to extend the network lifetime for a much longer period than the AMCA protocol, which already lost first connectivity after 15 days in scenario 5. In comparison GAF, was able to extend the lifetime to 173 days in scenario 5, which is a factor of almost 12 better than a node laying dormant in active mode without any radio activity.

Looking at tables 6.11 and 6.12, it is apparent that AMCA was able to route its messages with very little loss and only a few retries. Compared to the GAF protocol, very few routing messages on average had to be exchanged between the nodes to achieve this. This in itself is an indication that the implementation of the AMCA protocol is working well, but the protocol itself suffers from the problems that many other tree-based protocols also encounter. Namely the inability to distribute the load very well across the nodes in the network. Although AMCA takes fewer hops on average to deliver a message to the base station, it is often the same nodes that are routing the messages. The nodes that are closest to the base station has to stay active almost all the time, as they are often on the top of a tree branch that will become disconnected if it goes to sleep. This means that it is hard to share the load between nodes, as it becomes harder to route messages around

Description	S1 gaf	S1 amca	S2 gaf	S2 amca	S4 gaf	S4 amca
Network connectivity loss (days)	59,77	27,3	57,87	15,6	59,75	15,6
Average lifetime (days)	41,32	23,4	57,54	15,6	38,23	15,6
First node death (days)	31,04	23,4	57,38	15,6	31,04	15,6
Total unique messages lost (%)	0,12	0,14	0,83	0,22	0,91	0,00
Routing messages sent average	90	13,9	55	8	72	8
Messages routed on average	197	41	326	33	433	45
Retries on average	18	2,6	51	2,1	75	1,8
Avg. energy at con. loss (J)	3505	19251	8147	20755	7170	18207
Std. deviation of average (J)	7388	14616	7894	10000	7826	9961

Table 6.11: Results for low density scenarios.

Description	S3 gaf	S3 amca	S5 gaf	S5 amca
Network connectivity loss (days)	163,66	-	173,36	15,6
Average lifetime (days)	91,33	-	173,36	15,6
First node death (days)	46,17	-	173,35	15,6
Total unique messages lost (%)	0,73	0,42	2,5	0,67
Routing messages sent average	52	22	42	8
Messages routed on average	535	59	732	25
Retries on average	97	5	105	4,2
Avg. energy at con. loss (J)	10974	-	9941	27849
Std. deviation of average (J)	7835	-	7342	10773

Table 6.12: Results for high density scenarios.

nodes with little energy. It is a less flexible overlay network so to speak.

In a test setting such as the one we are using in our evaluation, the above problem will become even more apparent, as we are using very long and narrow topologies, resembling the structure of a bridge. In such a setting, a tree based routing protocol will lead to a very deep tree overlay network. This means that the nodes furthest away from the base station will be in a branch many levels down in the tree. Thus all these nodes are vulnerable if any of the current members of the tree in higher levels runs low on energy. Potentially, a large amount of nodes may become disconnected if a node close to the base station runs out of energy.

GAF on the other hand, depends almost exclusively on the density of the network. It is able to have the nodes close to the base station sleep, if they are low on energy. This means that the load in a given region of the network is shared across the nodes present in that region. It leads to longer average routes for messages, as they are typically sent a shorter distance each hop then in a tree based topology, but it makes GAF independent of longer chains of connected nodes, which is one of the vulnerabilities of AMCA as described above.

Having longer routes, however, comes with the price of increased retries and packet

loss but these are still kept at acceptable levels. Similarly, GAF sends out more routing messages on average than AMCA does, but both of the protocols scale well in respect to this number. Enlarging the topology does not seem to cause any significant increase in the amount of routing messages sent, and thus neither of the protocols will flood the network as the size of the network grows larger, which is important for the scalability of the protocols.

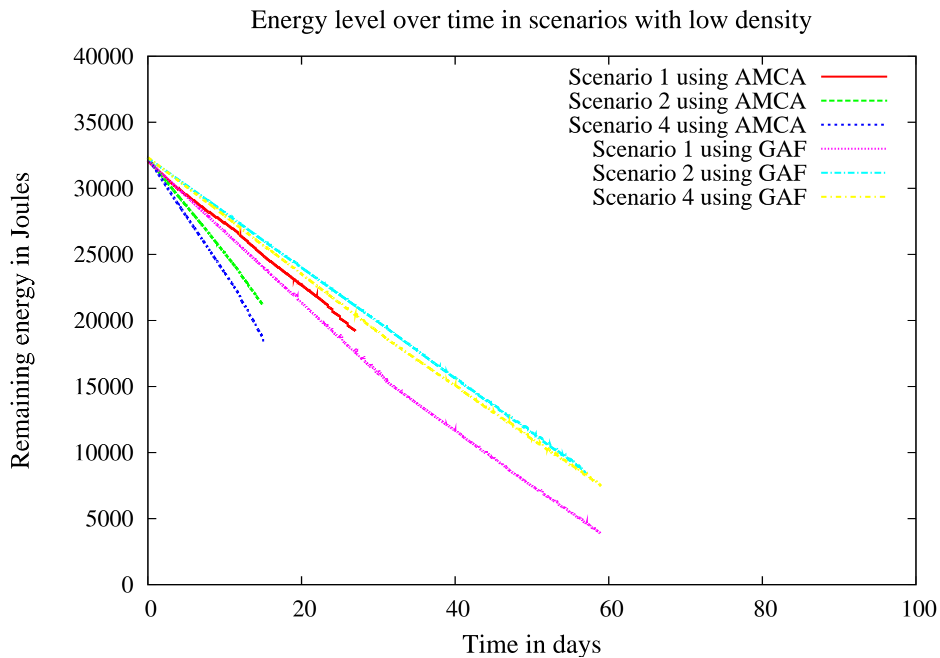


Figure 6.26: Energy level over time in scenarios with low density

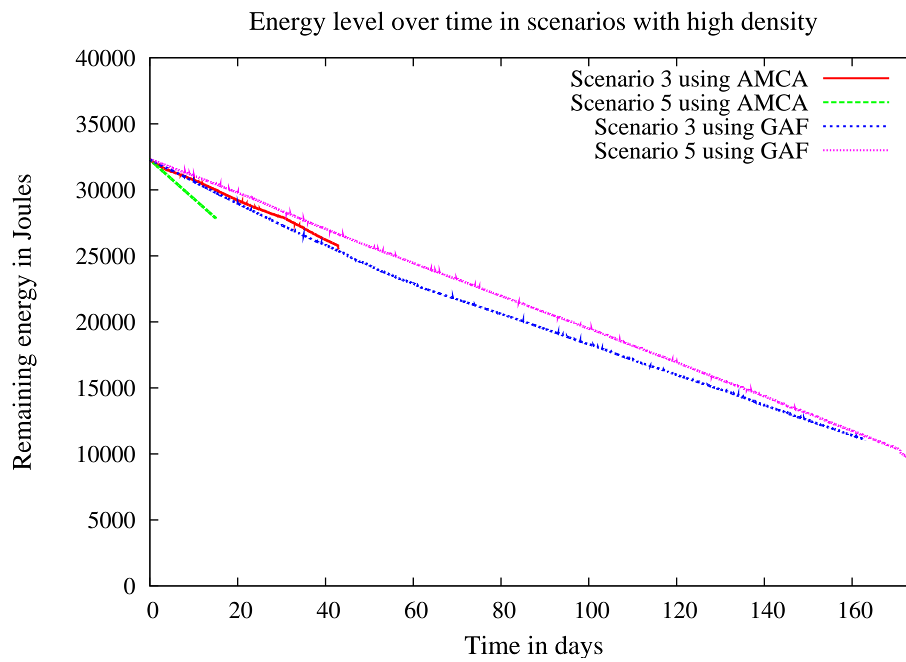


Figure 6.27: Energy level over time in scenarios with high density

7

Conclusion and Future Work

In this thesis we have implemented and evaluated two routing protocols: The *Geographic Adaptive Fidelity* (GAF) [16] protocol and the *Autonomous Multicast-tree Creation Algorithm* (AMCA)[15] protocol. The implementation of the protocols was done using TinyOS 2[5], which is an open-source operating system designed specifically for wireless sensor networks (WSNs).

The goal of the evaluation was to determine each of the protocols performance in a structural health monitoring (SHM) setting. In particular, we wanted to investigate the ability of each of the protocols to prolong the lifetime of a WSN in a SHM scenario. The danish research project *SensoByg*, which focuses on SHM, has been used and described, in an attempt to provide the most realistic testbed as possible for this evaluation.

In order to compare the two implementations, we have implemented a complete routing-layer framework which provides the same functional basis for both protocols. This means that the protocols can be evaluated specifically on their ability to perform as routing protocols, without having to worry about external variables such as message queues for the radio, and interaction with the application layer.

Each of the protocols were evaluated based on their performance in five settings, obtained from studying the preliminary requirements and results in the SensoByg project. The settings take their inspiration from a specific SensoByg scenario, concerning SHM in a bridge structure. This makes for very stretched rectangular topologies, which present the specific challenge for the protocols.

The tests were performed using the TOSSIM[12] simulator, which is a discrete event simulator specifically designed to simulate TinyOS applications. In order to use this simulator we had to develop our software for the MICAz[3] hardware platform which is a small sensor unit developed by Crossbow Technology[1].

As TOSSIM is not able to simulate the power consumption of the MICAz platform,

we also had to implement an energy model for our evaluation framework to use. The model is based on the hardware specifications for the MICAz platform, which lists the mA draw for different events and modes of the MICAz. It is thus a simplified model based on the power consumption of the platform under optimal conditions. Similarly, the decay of energy over time in a battery even if it is not used, has not been modeled. However, the energy model itself is sufficiently accurate for the purpose of evaluating the two routing protocols in the specified setting. For simulation purposes each node was given an amount of energy corresponding to 1 AA battery. We then investigated the lifetime that the protocols are able to achieve in each of the scenarios. For reference, a MICAz node which is given this amount of energy, can remain active for 15 days, if it does not send or receive any messages.

The results we obtained clearly show an advantage to the GAF protocol over the AMCA protocol in our SHM context. As explained in chapter 6, AMCA suffers from some inherent disadvantages due to its tree based nature. This makes it unable to scale well in the type of scenarios our tests was based on. GAF on the other hand appears to scale well with increased topology sizes. In fact, GAF is mostly dependent on the density of the deployed nodes. In the largest of our scenarios GAF achieved a lifetime of 173 days. Compared to the 15 days our reference node can stay alive, and given the positive results we obtained by simulating this protocol, we can conclude that the GAF protocol scales very well to large scenarios such as those found in many SHM settings.

However, as the Sensobyg project operates with desired lifetimes on a scale of 5 years and beyond, it is necessary to change one of two things for GAF to be truly applicable in this context. Either the density of the network must be increased, or each node must have access to a greater amount of energy. As the MICAz is built to use 2 AA batteries as default, the latter is of course an option. Though, doubling the amount of energy will not lead to a lifetime of 5 years or more. It is also possible attach an external battery pack, but this mean that each node will grow larger and will be harder to deploy. Increasing the density could mean having radios which are able to broadcast messages further than 10 meters while embedded in concrete, which preliminary results from the SensoByg project seems to be pessimistic of using current technology. It is also possible to simply deploy the nodes closer than one meter apart, but this comes at an increased cost of deployment.

A third option is to implement an algorithm capable of making the entire network enter sleep mode for extended periods of time. This algorithm could function on top of GAF or AMCA, but would require additional timing to realize. It should, however, be possible to implement this on top of the framework we have created, and could lead to significantly increases in network lifetime. This possibility is beyond the scope of this thesis and is left as future work.

7.1 Future Work

To introduce a more solid basis for our results, it could be beneficial to perform all our tests several times, and consider the average results and the confidence intervals. As explained, this was not possible in a realistic timeframe for this thesis, due to the poor

performance of TOSSIM when running the type of simulations we did.

Also, our initial goal was to run simulations with a thousand nodes or more. Again due to the performance of TOSSIM and the memory constraints induced, this was not possible. It would be interesting to investigate if the results we obtained concerning the scalability of each protocol will also be as apparent on an even larger scale. However, our results are still representative of the performance for each protocol, and thus our conclusions upon their scalability should still hold when performing simulations on a larger scale.

7.1.1 GAF

Although we have made a few optimizations to improve the performance of GAF and described some possible solutions to the problems we discovered, there are still ideas beyond the scope of this thesis that could be considered. One of these concerns a more adaptive manner of creating the grid cells as described in [9].

There is also room for improvement in the geographic routing used by GAF to forward messages to the base station. The simple implementation that was done, is not able to recover if a message is routed into a dead end, as it will never route a message back to where it came from. Solutions to this problem have been proposed, and more advanced geographic routing protocols such as [11] are able to cope with such problems. This could lead to a longer lifetime as some grids could potentially be able to route their messages through more advanced routes, and thus lose connectivity to the base station later.

7.1.2 AMCA

The biggest issue discovered with the AMCA protocol during the simulations, was the fact that in many of the simulations, the network lost connectivity at 15,6 days. This means that some nodes never went to sleep, which is one of the main points of this routing protocol.

One way to fix that problem, is to make it so that if a node can only reach very few nodes in the level above, we instead just move the node one level down. That would mean that the node could reach many more nodes in its previous level since they are closer but it would also mean that a change to the level mechanism would have to be made since the leveling is based on the identity number of the node as described in section 3.2.1. That could be fixed by disabling the identity based leveling mechanism after the initial tree creation. That would mean that after the initial tree has been created, a node could not move up or down a level, just based on its identity number.

Another way to alleviate this problem, is to move the node one level down after having lost connection to all nodes in the level above. This would mean that some nodes would die after having never gone to sleep, but the nodes that were dependent on them would still find a way to route their data messages to the base station. The first solution is probably the better solution since that would allow all the nodes to opportunity sleep, hence increasing the lifetime of all the nodes in the network.

Bibliography

[1] Crossbow technology. http://www.xbow.com/.

[2] Duracell AA battery data sheet. http://www.duracell.com/oem/Pdf/new/MX1500_US_UL.pdf.

[3] Micaz data sheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf.

[4] Sensobyg project. http://www.sensobyg.dk/.

[5] Tinyos. http://www.tinyos.net/.

[6] Saurabh Ganeriwal, Deepak Ganesanl, Mark Hansen, Mani B. Srivastava, and Deborah Estrin. Rate-adaptive time synchronization for long-lived sensor networks. In *SIGMETRICS '05: Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 374–375, New York, NY, USA, 2005. ACM.

[7] Wendi Rabiner Heinzelman, Anantha Chandrakasan, and Hari Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8020, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185, New York, NY, USA, 1999. ACM.

[9] T. Inagaki and S. Ishihara. A scheme for expanding grid size of geographical adaptive fidelity. *Networked Sensing Systems, 2007. INSS '07. Fourth International Conference on*, pages 291–291, June 2007.

[10] Peng Jiang, Yu Wen, Jianzhong Wang, Xingfa Shen, and Anke Xue. A study of routing protocols in wireless sensor networks. *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, 1:266–270, 2006.

[11] Brad Karp and H. T. Kung. Gpsr: greedy perimeter stateless routing for wireless networks. In *MobiCom '00: Proceedings of the 6th annual international conference*

on Mobile computing and networking, pages 243–254, New York, NY, USA, 2000. ACM.

[12] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[13] S. Lindsey and C.S. Raghavendra. Pegasis: Power-efficient gathering in sensor information systems. *Aerospace Conference Proceedings, 2002. IEEE*, 3:3–1125–3–1130 vol.3, 2002.

[14] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, New York, NY, USA, 2004. ACM.

[15] Katsumi Onodera and Toshiaki Miyazaki. An autonomous multicast-tree creation algorithm for wireless sensor networks. *Future Generation Communication and Networking*, 1:268–273, 6-8 Dec. 2007.

[16] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 70–84, New York, NY, USA, 2001. ACM.

[17] Hugh D. Young and Roger A. Freedman. *Sears and Zemansky's University Physics, 11th edition.* Pearson Addison Wesley, 2004.

[18] Y.Yu, D. Estrin, and R. Govindan. Geographical and energy aware routing: a recursive data dissemination protocol for wireless sensor networks. *UCLA Computer Science Department Technical Report*, UCLA-CSD TR-01-0023, May 2001.