THE LUSUS PROTOCOL

A THESIS SUBMITTED TO THE GRADUATE DIVISION OF THE
UNIVERSITY OF HAWAI'I IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

INFORMATION AND COMPUTER SCIENCES

AUGUST 2005

By
Daniel H. Morton

Thesis Committee:

Edoardo S. Biagioni, Chairperson
Kim Binsted
Matthew McGranaghan

For my family,

Ron, Kathy, Amy, John, and lil Sissy,

for always being there when I really needed them.

Acknowledgements

First and foremost I would like to thank my committee, Edo, Matt, and Kim, for all their help and guidance throughout the entire process of my thesis. It's been a long journey full of lessons, some of which were learned the hard way, and all of which have made me a better computer scientist and person.

I'd also like to acknowledge the contributions of Brian Chee and Gerard Fryer, without which Lusus would not exist. Many thanks go to the Advanced Network Computing Laboratory and the Laboratory for Interactive Learning Technologies for the use of their equipment and resources. I'd also like to thank the two labs for employing me and allowing me to pay for school.

Last but certainly not least I would like to acknowledge and thank my parents for the excellent upbringing they have provided me. Without their encouragement, love, and support I would never have been able to make it this far in academia.

ABSTRACT

Wireless sensor networks are groups of nodes which sample data from one or more attached sensors and cooperate via wireless links to transmit this data to a destination. This document introduces Lusus, a new protocol designed to operate wireless sensor networks for ecological monitoring. Unlike other protocols with a more general design focus, Lusus assumes that the vast majority of information travels towards a central point. This allows Lusus to use specific routes in an efficient manner since the only route a node need know is the next hop towards the center of the network. Lusus uses a limited form of route discovery transmitted periodically from the center of the network and relayed by each node in the network. This periodic route discovery flood is done on the order of hours to save bandwidth. The routing overhead in Lusus is significantly less than in other protocols.

Lusus assumes that data items are small (several bytes) in size. Furthermore Lusus is designed to to optimize transferring small units of data. Individual pieces of data in a Lusus network are packaged within self-contained units. Because of this, nodes are allowed to combine the data from multiple separate packets into a single outgoing packet. This allows Lusus networks to save on overhead and thus increase their efficiency. This combining of data results in an overhead of 27% per piece of data whereas without combining the overhead is 85%. To help ensure data reception, Lusus uses hop-by-hop acknowledgments. This type of acknowledgment is necessary to support the data combining feature of Lusus.

This document describes the operation of Lusus and offers an analysis of its performance for large and for dense networks.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

A wireless ad-hoc protocol controls communication in an unstructured, decentralized network of wireless nodes. These protocols regulate how, when, and along what path data is transmitted. There is no hierarchy; all nodes are assumed to be homogeneous. As part of this there is no central authority that has an overview of the network.

Everything from factory automation to botanical field studies can make use of these protocols. Traditionally small, low-powered computers (microcontrollers) have been used as the hardware platform of choice. These microcontrollers have dictated that the protocols be simple and relatively lightweight with respect to implementation. Furthermore, microcontrollers are usually powered by some combination of batteries, solar panels or capacitors, necessitating that the nodes "sleep" a good percentage of the time to save power. To say that nodes "sleep" is to say that they periodically go into a low-power state to conserve energy. The downside to this low-power state is that peripherals such as the radio transceiver are switched off.

Wireless ad-hoc protocols must be able to gracefully handle changes in network topology. Given the size and weight of the nodes in such a network, one must assume that they will move about from time to time. Also, the ability to move is an asset in some kinds of monitoring.

Nodes in wireless ad-hoc networks must cooperate with their neighbors to establish routes to specific destinations. In this way these networks use a peer-to-peer methodology. Nodes depend on other nodes to establish connectivity.

Wireless networks tend to have a significantly higher bit error rate than their wired counterparts. This is primarily due to the higher number of possible error sources in wireless networks. These sources include all matter of electromagnetic interference, physical objects impeding carrier waves, and especially collisions among packets. Furthermore, wireless networks tend to completely lose packets from time to time [20]. Packets are lost in wireless networks in part due to the same sources which produce errors. However, to completely block a packet from being received, the error sources must occur for the entire duration of packet transmission. These two factors make error detection and handling a non-trivial exercise for wireless ad-hoc networks.

1.2 Problems Addressed

In the most general terms wireless ad-hoc protocols help to solve the problem of distributed communication. The communication is distributed in that it proceeds at the discretion of the nodes involved. There is no central authority regulating communication in this kind of network. These protocols enable nodes to use their neighbors as relays for messages. This in part helps to solve the problem of nodes who wish to communicate but are not within radio range of one another.

In specific terms, Lusus helps to solve the problem of distributed sensing by enabling communication between sensor nodes. Distributed sensing provides additional pieces of data from different locations. Communication in these networks ranges from relaying of data to the coordination of sensing efforts. This coordination of sensing efforts provides a better picture of the phenomenon (i.e. weather) under study.

1.3 COCONuts Project

The COCONuts project was a design study for a network of communications sensors for tsunami early warning [12]. This network was designed to consist of coconut

sized computer systems floating throughout the Pacific Ocean. These systems would use GPS to determine their position, and to determine if a tsunami had passed beneath them. In the event that one had, the floating computers would signal operators throughout the Pacific so they could take appropriate action.

Equipping each node with a satellite link would have been prohibitively expensive, so there was need for a simple communication protocol that would allow several units to share a single satellite connection. This presented several interesting problems that other protocols have never had to consider, such as the constant movement of nodes.

The COCONuts project gave Lusus its first set of design assumptions. Further impetus came from the desire to use Lusus in another wireless environmental sensing project, the PODS project.

1.4 PODS Project

The PODS project is a multi-year DARPA-funded research effort into microsensor networks for ecological monitoring [3]. The ultimate goal of the project is to come up with inexpensive software/hardware combinations that can be used to monitor endangered species in Hawaii. As part of this the researchers need to take certain measurements periodically and have them relayed back to a central database for storage [2]. Most of these measurements are relatively small: temperature, humidity, light level. They don't require a large or complex system to take them or transmit them.

Lusus was adapted to gather the simple measurements needed by PODS. Nodes in a PODS network do not move all that much. However, the frequent "sleeping" of nodes does in some way introduce some of the same problems as the highly mobile COCONuts scenario. A problem such as frequently changing network topology is an example of this. When a node sleeps it is effectively out of communication with the network for however long the sleep period is.

1.5 Other Applications of Wireless Sensor Networks

Wireless sensor networks are either in use or proposed for use in a wide variety of situations. There is of course the aforementioned ecological monitoring, which was given a large boost by DARPA funding [8]. DARPA has also focused on homeland security applications.

Other applications of the technology are: residence monitoring, medical monitoring, and vehicle monitoring [7]. Residence monitoring includes the so-called "smart house" family of technologies. This form of monitoring can help keep track of and make adjustments to residential features (thermostat, lighting). Medical monitoring sensor networks can be used to keep track of patient vital signs in a hospital setting. These networks of small devices can enable more accurate monitoring of the status of a patient. Finally vehicle monitoring can be used in racing situations. These situations require a large amount of telemetry data to finely tune the car for the track conditions at that moment in time. In any of these situations wireless sensor networks can be used to provide more information for more informed decision making.

1.6 Related work

This section presents information on three protocols comparable to Lusus. All three serve to illustrate other methods of controlling communication in wireless sensor networks.

1.6.1 Flooding

Flooding is perhaps the simplest form of a wireless ad-hoc protocol. When flooding, a source node broadcasts a message to all the other nodes within its transmission range. Each of these nodes in turn rebroadcasts this message to their respective neighbors. This pattern continues until every node in the network has transmitted the message. Nodes can only rebroadcast a given message once, and different mechanisms can be used to achieve this. For example when Lusus uses flooding, it uses hop counts to make sure each node rebroadcasts each message exactly once. The simplest form of flooding results in colli-

sions and lost packets as nodes within range of each other then retransmit at almost the same time. To prevent this a random jitter can be introduced so that with a high likelihood the nodes are not transmitting the packet at the same time. Like Lusus, flooding is a very simple protocol. However, the performance of Lusus for wireless sensor networks is much better than the performance of flooding. This is discussed in detail in Section 5.6.

1.6.2 Gradient Routing

Gradient Routing (GRAd) is a mechanism for data routing and control in wireless ad-hoc networks [23]. GRAd uses a multi-hop architecture to provide communication between two nodes not in range of each other. This is done by way of intermediate nodes relaying messages. GRAd is an on-demand protocol – routes are formed only when nodes wish to communicate with each other. No attempt is made to maintain state when there is no data to send.

In GRAd nodes maintain a table of the relative cost to a destination. This table contains entries for each destination that a node knows about. The cost table in GRAd is analogous to the route table in other protocols. Both contain information about where to forward packets to based upon their destination address. Whenever a node receives a message it searches the cost table for a corresponding entry. If no entry is found, the node creates a new entry with the cost of the sender, based upon the information in the message. If an entry is found the node must compare the sequence number as presented in the message with the sequence number in the entry. If the sequence number in the message is greater than the one in the table, the table entry is updated. Otherwise the message is marked as a duplicate.

A node wishing to transmit to a destination does not attempt to identify which neighbor is to relay a packet. It merely broadcasts the packet and each node that can relay it at a lower cost will relay it. The result of this is that packets follow the steepest gradient to the destination, where the difference in costs among nodes is the steepness of the gradient between them. In this way routing in GRAd is dynamic and does not use a specific neighbor for message relaying. In order to reduce latencies GRAd does not use an

RTS/CTS [1] medium reservation mechanism. This reduces latency by forgoing the hop-by-hop transmission of RTS/CTS packets.

1.6.3 Directed Diffusion

Directed Diffusion (diffusion) is a method of "coordinating the distributed sensing of environmental phenomena" [19]. In diffusion all sensing is done as a result of a request for a specific piece of information. This request for information (also known as a task) is disseminated throughout the network using gradients similar to those in GRAd. These gradients are also used to return the requested information to the source of the request.

All nodes are assumed equal and support each other for task and answer relaying. Diffusion will disseminate a task throughout the network and reinforce an optimal path (or paths) between the sensing nodes and the requesting node. Once these paths are established the requesting node will periodically broadcast a "reminder" packet of sorts. This refresh packet will keep the gradient alive as well as keeping the sensing node on the job.

1.7 Design Assumptions for Lusus

Purpose Lusus was designed to provide connectivity for networks of small, embedded computers communicating over wireless links.

Communication Direction Communication is assumed to be towards a central point of data storage. In wireless sensor networks the objective is to sample data throughout the network and relay it back to a central point (a base station) for presentation to consumers of the information.

Communication Range The nodes are assumed to not all be within range of the central base station. This is because of limitations on radio range; it is not always feasible or even possible to have radio contact between every node and the base station. Lusus uses other intermediate nodes to act as communication relays.

[1] Request To Send / Clear To Send: a method to avoid collisions in wireless networks such as IEEE 802.11 (WiFi).

Remote Control The nodes are assumed to need little in the way of instruction or control. Nodes are very simple devices as far as computers go. They are designed to sample data and transmit it. These two functions are programmed in during assembly and tend not to change. If any part of the function does change, it is usually the rate at which samples are taken. This is not a difficult change and can be easily transmitted to all the nodes in the network from the base station, using flooding to communicate the change to all the nodes.

Topology The topology of a Lusus network is assumed to be dynamic. This is assumed to be the result of either node movement or node status (nodes going on and off line). Node movement (as in [12]) changes the physical topology of the network. Node status (as in [3]) changes the logical topology rather than the physical topology of a network. It is furthermore assumed that a combination of movement and status changes can occur in the same network.

1.8 Design Methodology

Lusus was designed using a three-stage methodology. The first stage of this methodology has the problems to be solved pondered and initial concepts determined. For example, Base Station Announcements (Section 2.3) were developed during this stage. The second half of this stage has the ideas and concepts put to paper. Any gaps or inconsistencies which become apparent are remedied with additional details or revision. This stage leaves the protocol as a relatively complete specification document, but with no validation to prove it will work.

Following the initial design stage, a protocol is implemented in some way. This implementation can take the form of a simulation or program running on real hardware. The purpose of implementation is to discover any issues in the protocol that were not detected during the design stage. These issues are in turn fixed by changes to the protocol. The implementation has another purpose in addition to issue discovery: validation. An implementation shows a protocol operating. This is in essence a proof of concept showing

that the protocol will work. In most cases [6] [23] protocols are, at least initially, simulated. However, Lusus was implemented on real hardware.

The third and final stage of this methodology is evaluation. The process of evaluation for Lusus provides results in the form of analysis and mathematical formulas. These results are useful for determining the performance of a protocol under conditions different from those in the implementation. The results also aid comparisons with other pre-existing protocols. Finally the results from evaluation offer further evidence that the protocol in question will function as designed.

This document describes all three stages with respect to Lusus. It gives the details of each stage as well as the results from the evaluation. Finally it provides some discussion of the results.

1.9 Sample Scenarios

1.9.1 Ecological Monitoring

Consider a network of nodes running Lusus deployed to study the endangered plant species *Silene Hawaiiensis* (Figure 1.1). This sample scenario is inspired by an actual field study carried out by [3]. Using Lusus nodes means that the researchers do not have to walk to the study site several times per day to gather weather samples. Also using Lusus nodes would mean less foot traffic in the study area. This will help to lessen the impact on the fragile *Silene*. These nodes would need to sample temperature, light, humidity, wind, and rainfall levels. There is a research station approximately 1km away from a sizable group of plants. This research station contains power and Internet connections suitable for a base station. Small nodes with the relevant sensors and a radio transceiver could be constructed easily and with relatively low expense. Some of these nodes would then be placed in a rough line along the 1km stretch. The balance of the nodes would be scattered around the field of plants. Due to the low cost of the Lusus nodes this deployment can afford to scatter them about. This is because there will be a sufficiently high density of nodes throughout the network which will ensure connectivity. Finally the nodes should, upon completion of the study, be collected for recycling or reuse in another study.

Figure 1.1: *Silene Hawaiiensis*. Photo Courtesy K.W. Bridges

1.9.2 Collaborative Forest Survey

In this scenario nodes [5] running Lusus are "connected" to a network of another type of node communicating using the Multi-path On-demand Routing protocol (MOR) [6]. The MOR nodes provide higher-bandwidth (11Mbps) connections. These connections are used to carry larger pieces of data (i.e. high-resolution photographs) in addition to data from Lusus nodes. In this network the MOR nodes form a backbone of sorts. The Lusus nodes are deployed in a three-dimensional fashion around a specific study area. A single MOR node functions as a base station for a local group of Lusus nodes. Each base station uses MOR to relay the data from a group of Lusus nodes to a central point of storage.

1.10 Chapter Summary

This chapter has introduced many of the concepts behind wireless ad-hoc protocols and wireless sensor networks. It has provided some examples of the possible uses of wireless sensor networks. A few different protocols for wireless ad-hoc networks were described and compared to Lusus. The chapter also has given some insight into what general problems are solved by wireless ad-hoc networks. Finally this chapter has given details of the methodology of how Lusus was designed. The rest of the thesis is structured roughly along the same lines as the design methodology: in Chapter 2 a specification of the protocol, in Chapter 3 a description of the protocol implementation, in Chapter 4 an analysis of Lusus performance, and in Chapter 5, 6, and 7 a discussion of the results, future work, and conclusions.

Chapter 2

Lusus Protocol Specification

2.1 Introduction

Presented in this chapter are the workings of Lusus in general terms. This chapter is not meant to be a full guide to implementation, but rather an introduction to how Lusus works.

Additional low-level details useful for a Lusus implementation are given in Appendix A. This includes specific details and constants chosen for this implementation of Lusus.

2.2 Datagrams

A Datagram is a fixed-size data field along with some information to uniquely identify the piece of data. Some of the identification information is implementation-dependent. For example a datagram containing a barometric pressure value must be assigned a unit of measurement before it is of much use. Which units to apply are known *a priori* because they are based upon the type of sensor which recorded the value.

Tables 2.1 through 2.3 show conceptual examples of datagrams. Table 2.1 contains a temperature value of 25 degrees celsius recorded on 4/3/2005 at 16:45:28. Table 2.2 gives an example of a light level reading of 20% of full perpendicular solar illumination taken on 4/3/2005 at 16:45:28. Finally, Table 2.3 shows an example humidity reading of 50% sampled on 4/3/2005 at 16:45:28.

Table 2.1: Temperature Datagram

temperature	4/3/2005 16:45:28	25

Table 2.2: Light Level Datagram

lightlevel	4/3/2005 16:45:28	200

Table 2.3: Humidity Datagram

humidity	4/3/2005 16:45:28	50

2.3 Packet Types

Base Station Announcement Base Station Announcements (BSAs) are used to announce the presence of a base station. They contain a single datagram with information for the outlying nodes. These packets are primarily used to form the routes necessary for Lusus to function. They are also used to provide some form of clock synchronization throughout the network. Base station announcements usually carry one or more datagrams.

Data Packet Data packets are used to return sensor data back to the base station. They depend on intermediate nodes to relay these packets until they reach the base station. Data packets contain one or more datagrams, possibly from different nodes.

Data Acknowledgment Data Acknowledgments are used to acknowledge the successful receipt of a data return packet. These packets contain a single datagram carrying a signature based upon the packet being acknowledged.

Node Command Node Command packets are used to give instructions to specific nodes in a Lusus network.

Network Diagnostic Packets The following two types of packets are used to help with configuration and troubleshooting of Lusus networks.

Network Ping These packets are used to determine if a specific node is online.

Network Traceroute These packets are used to determine the routes within a Lusus network.

2.4 Theory of Operation

Route Formation Routes in Lusus are formed by the relaying of Base Station Announcements (BSAs). Initially a base station will send out a BSA addressed to the broadcast address. The data field within the single datagram is used to keep track of the number of times the packet has been relayed. When initially sent from the base station the data field is set to zero. Each time a node receives a BSA it must first decide whether or not to retransmit the packet. This decision is based upon the hop count transmitted in the packet. If the hop count is less than the node's current routing distance to the base station the packet is retransmitted. If the packet is to be retransmitted the node must update two pieces of information within the packet before retransmission, specifically the hop count in the data field, and the ID of the node transmitting the packet.

Persistent Routes Nodes in a Lusus network use *persistent* routes to deliver packets. Once a node has a route to the base station it remembers this route for use in routing future packets. Lusus uses these persistent routes by specifically storing the address of the next hop towards the base station in memory.

Loss of optimal route In Lusus, as in any wireless network, nodes will come in and out of contact with the network. This is due to a number of factors, for example power failure, hardware failure, environmental conditions, low-power sleep mode, movement out of range, or other objects interfering with transmission. It is entirely possible that the shortest route to the base station may experience a node failure, and thus be unusable. In this case nodes must use a more costly route (in terms of hops) to the base station. This presents a problem in that Lusus nodes will ignore any routes that are longer than their current one. To solve this problem Lusus nodes must periodically flush their route table. This flushing of the route table allows the node to properly evaluate BSAs that it hears and rebuild its routing table accordingly. Implementers of Lusus are free to vary the time

between route table flushing. Networks in which the effective topology changes rapidly should have a short time between route table flushes. Conversely, networks with a relatively static configuration should maintain their route tables for longer periods of time.

Data Return Nodes transmit data back towards the base station with data packets. These data packets contain one or more datagrams. Lusus does not specify a minimum transmission threshold, so nodes can transmit packets with as little as one datagram. However, as seen in Section 4.4, it is beneficial to queue a few datagrams before sending a data packet.

Data Combining Since the format of a data return packet is modular, Lusus allows intermediate nodes to add datagrams to a data return packet they are relaying.

Acknowledgment At each hop along a route, reception of data packets is acknowledged. When a node receives a data packet and has verified that the packet was received successfully it must acknowledge the original data packet. The acknowledging node creates a new packet. The header of this new packet must be addressed to the most recent sender and must identify the packet as an acknowledgment. To this new packet the node must copy the integrity signature (CRC) of the original packet. Finally the packet is transmitted. The recipient of the acknowledgment can then remove the datagram from its retransmission queue.

Retransmission In Lusus, nodes are required to retransmit unacknowledged data packets at certain intervals for a certain amount of time. Retransmission in Lusus is designed to increase the probability that data is received by the base station. In order to keep the network from flooding with retransmission traffic, Lusus uses a binary exponential backoff algorithm for timing the retransmission of packets. So if the first retransmission occurs two seconds after the initial transmission, the next retransmission will occur four seconds later. In this scheme nodes will try to retransmit packets forever. This uses up valuable resources. To avoid this, Lusus sets a maximum number of retransmissions per packet. If the packet is still not acknowledged by the time a node has retransmitted it the maximum number of times, it is simply discarded.

Integrity Signature Lusus uses a 32-bit Cyclic Redundancy Check (CRC) to help maintain the integrity of packets. A CRC is computed for and included in each and every packet transmitted in a Lusus network. This CRC forms the basis for detecting errors within a packet. When a packet is received, this is the second thing a node must check during the processing of the packet. To check the CRC a node computes the CRC of the entire received packet (including the original transmitted CRC). If the computed value is not zero, the packet must be discarded.

Packet Framing In order to determine where one packet ends and another one begins Lusus uses specific byte sequences to signal the beginning and end of packets. These specific byte sequences are sent in addition to the rest of the packet. However, it is entirely possible that the specific byte sequences will occur within the body of the packet. In order to avoid ambiguity, Lusus requires another byte sequence be prefixed to every instance of the start and end sequences that occur in the body of the packet. In short terms Lusus uses the SLIP[1] protocol [24] for packet framing.

Network Ping Network pings are assumed to originate from a base station. They are used to determine how many hops there are to each node on the network. To process a ping request packet a node must do three things. First, if the hop count of a received packet is not equal to the current one in memory, the node must discard the packet. Next, the node will forward the packet along having changed the source address to its own and incremented the hop count by one. Finally it will compose a ping reply packet. A ping reply packet is very similar to a data packet. It includes a single datagram used to keep track of the route length. When a node receives a ping reply packet it must forward it along towards the base station.

Network Traceroute Network traceroutes are very similar to network pings. They are used to determine the routing paths in use in the network. They must originate from a base station. On receiving a traceroute request packet a node is required to do two things. First it is to forward the packet along having changed the source address to its own and

[1] Serial Line Internet Protocol: a method of transmitting IP packets over serial lines.

incremented the hop count by one. Second it will compose a traceroute reply packet. This traceroute reply packet contains a single datagram containing the node's distance from the base station in hops. When a node receives a traceroute reply packet it must add a new datagram at the end of the packet. This datagram contains the relay node's address and hop count.

Command Packet Command packets provide a limited amount of control over nodes in a Lusus network. These packets contain at least two datagrams: one to keep track of the number of hops a packet has traveled and one or more containing the actual command information. This command information is highly implementation-dependent. It can range from simple scalar values to actual program code. Regardless of how it is implemented, command packets allow repurposing of nodes in a Lusus network.

Urgent Data There exist cases where certain pieces of data transmitted by a Lusus network are of a more time-sensitive nature than the other pieces of data. To help address this issue Lusus has support for urgent data packets. These packets (denoted by a special flag) are special in that they are not allowed to be combined with other data packets. When a node receives an urgent data packet it must, as immediately as possible, relay the packet without adding any additional data. In this way the urgent packet carries a higher priority for relay to the base station. Urgent data packets can be inefficient due to little or no data combining, but this inefficiency is irrelevant if the data really must be transmitted urgently.

2.5 Lusus Medium Access Control

Lusus uses a custom Medium Access Control (MAC) mechanism. This custom MAC is used because it results in low overhead and low latencies. Furthermore it creates a small block of code on the target device. The Lusus MAC is what controls how Lusus nodes share a common communications medium. As is common in wireless communications, Lusus assumes that only one node at a time can transmit within a given area. If two Lusus nodes within range try to transmit at the same time the result is a random combination of

the two original transmissions which will with very high likelihood carry an incorrect CRC and packet length.

The Lusus MAC is a variant on CSMA[2]. This MAC mechanism is a very simple, passive way to avoid some percentage of collisions. Lusus nodes must wait until no other node within their reception range is transmitting before they begin to transmit.

2.6 Chapter Summary

In this chapter the operation of Lusus was discussed in general terms. The specific, numeric details were not included for clarity. Also, it is possible that these numeric values will change over time, but the information presented in this chapter will most likely not. The information presented in this chapter should give a clear understanding of how Lusus works. The major operations performed by the protocol were all presented. As evidenced by the relatively small size of this chapter Lusus is a relatively simple protocol. For the target applications a simple protocol is the ideal.

The next chapter describes the implementation of Lusus.

[2]Carrier Sense Multiple Access: A MAC mechanism that waits for the carrier to be idle before transmitting.

Chapter 3

Protocol Implementation

3.1 Introduction

As part of its development and evaluation Lusus was implemented on a small scale. This implementation gave hard numbers and validation of the performance of Lusus. In this chapter some detail of the implementation is given. Also, some of the issues and solutions, if available, are presented. This helps to give some insight as to why certain features of Lusus are the way they are. The complete code of this implementation forms Appendix B. The chapter concludes with a discussion of hardware which could be used in a field-deployed Lusus network.

3.2 Why Implementation?

In most cases protocols go from design to simulation then implementation. However, Lusus went from design straight to implementation. This occurred for a number of reasons. First and foremost, Lusus is inherently a simple protocol. An implementation would also be a relatively simple program not requiring much in the way of development time (in comparison to other, more complex protocols like 802.11). Simulators are complex pieces of software. It would have taken more time to create a Lusus module for a network simulator than it did to create an implementation from scratch. Finally the PODS project (which helped create Lusus) had sufficient available equipment to make an implementation practical.

It is important to note that many other protocols, when presented to the scientific community, exist only as simulation code. Simulators are not infallible. Indeed, recent surveys of published papers in the field of wireless ad-hoc networks suggest that there are many problems with simulations, simulators, and the way the results are presented [21]. The results of a simulation are only as good as the program written to run them. The research presented in [21] has also found that different versions of the popular network simulator *ns-2* [18] give different results when presented with the same input.

3.3 Implementation Details

The Lusus implementation uses Personal Computers (PCs) running the Linux operating system as nodes. These nodes use 9600 bps serial transceivers to communicate. These transceivers use omnidirectional antennas. The nodes are arranged logically in a straight line so that each node is only within range of the node before it and the node after it (Figure 3.1).

The software, written in the C programming language, allows any node to act as either a base station or regular node. This allows for the evaluation of multiple different network topologies without a redeployment of the nodes. Furthermore, the use of C makes it easier to port the software to different hardware platforms.

The first result obtained from the testing implementation is that Lusus works as expected. Each node is able to build a routing table based upon the relaying of BSAs from the base station. Nodes are also able to transmit simulated data back to the base station. This is accomplished by both straight relaying of packets and data combining. When performing data combining there is an observable decrease in overall network traffic.

Network traceroutes are implemented. Network traceroutes list the nodes along the routes that exist within a Lusus network. This list as presented in a packet is in reverse order; the first node listed is actually the furthest one from the base station. They have proved quite useful for configuration and troubleshooting of the network. Table 3.1 shows an example traceroute from the test network. A diagram of the test network is shown in Figure 3.1. For node addresses assume that topmost node is the base station. Also assume that nodes A, B, and C are the remaining three nodes in a clockwise fashion.

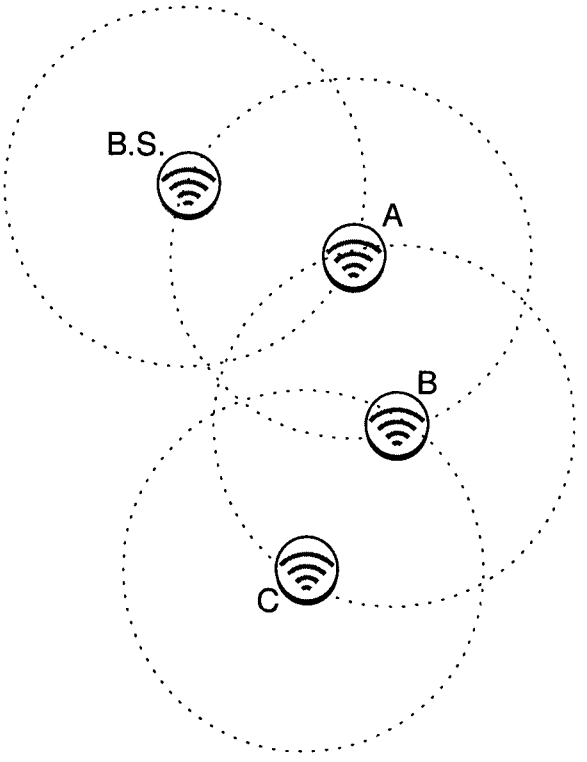

Figure 3.1: Diagram of Test Network

traceroute	4/20/2004 8:05:28	0x000C
traceroute	4/20/2004 8:05:40	0x000B
traceroute	4/20/2004 8:05:53	0x000A

Table 3.1: Traceroute Datagrams

3.4 Node Placement

The manner in which nodes were placed proved to be of critical importance during the deployment of the testing implementation. As in Figure 3.1 each node must be within communications range of at least one other node. Wireless communication is significantly more complex than wired communication. Signals from a wireless transmitter travel away from the antenna in all directions (in the case of an omnidirectional antenna). These radio waves are deflected or absorbed by objects depending on the physical properties of the object. This process of deflection and absorption can make placement of wireless equipment more challenging that for wired networks. The testing implementation is no exception.

When deploying and testing the implementation, great care had to be taken to place the nodes to ensure bidirectional communication. The best solution was to underestimate the transmission range of the nodes. In underestimating the transmission range the nodes could be placed in such a way as to increase the average signal-to-noise ratio[1] to help ensure connectivity.

In a larger scale deployment the issue of fluctuations in connectivity can be dealt with by increasing the node density, which will increase the number of possible communication links. This increase in possible communication links will lead to an increase in the probability of all nodes being able to communicate with at least some of their neighbors at any given time [14].

[1]Signal-to-noise ratio is the ratio of signal strength vs channel noise. It is useful for determining the effective signal strength at a point in space

3.5 CSMA

The testing implementation did not make use of CSMA. While creating the implementation it was discovered that the particular pieces of hardware and software being used would make inclusion of CSMA a non-trivial exercise. Specifically the serial radio devices in use buffered a few bytes of data before sending it to the attached node. This buffering made it difficult to tell if another node was actually transmitting at that time. It was decided to forgo CSMA in this particular version of the Lusus MAC. This decision reduced the performance of the testing implementation significantly. Without knowing when other nodes were transmitting each node had to use a longer random jitter in their packet transmission time. This jitter was on the order of seconds. Without this jitter, collisions in the very small test network quickly became a large problem. Tests could still be run, but the performance suffered. As such the primary lesson learned is that including CSMA in any implementation of Lusus is of great importance. Without it, the performance is degraded significantly either through collisions or jitter used to help prevent them. Without anything to help avoid collisions greater than 75% of packets were lost. When using jitter packet loss dropped to less than 10% but latency increased to several seconds.

3.6 Node Synchronization

In wireless sensor networks synchronization is important. It is this synchronization that allows networks of nodes to communicate even though the nodes spend a significant portion of their time in low-power mode. This limits the time available for transmitting packets. As nodes try to send more often, the likelihood of collisions increases.

The testing implementation of Lusus has showed that synchronization between nodes can significantly increase collisions. This implementation does not use nodes that "sleep". However, the software implements a delay loop which approximates this sleep behavior. In the first version of the software, the nodes would sample their data and transmit it once per delay loop. This led to an inordinate number of collisions.

After some analysis of the problem, the software was changed to add a slight jitter to packet transmission. Nodes wait a random amount of time before actually sending

their data. This jitter reduces collisions. By reducing collisions more packets are ultimately received at the base station.

3.7 Platform Evaluation

As part of further refining Lusus for an actual implementation, several different types of microcontrollers were evaluated and tested. The optimal target would have a large amount of RAM (for a microcontroller), almost 4 kilobytes. This large amount of RAM is necessary for packet queuing. Also the optimal system would be programmable in both C and assembler. The code necessary to support Lusus would be large and complicated in relation to the average microcontroller program. Being able to use C would ease the development cycle resulting in a better product in less time. The optimal system would have to have at least one serial port. The serial port is used to communicate with the radio device. Finally the optimal system would have several Analog-to-Digital (A/D) interfaces. These A/D interfaces would be used to read values from standard resistive sensors. Some of the systems evaluated include PIC controllers from Microchip, some of which offer all the desired functionality.

3.8 Lessons Learned

Before settling on Linux-based PCs, an embedded solution was first considered (Section 3.7) to serve as the hardware basis for the testing implementation. Even with a common programming language (C), programming for Linux is far different from programming for embedded devices. For example, in the type of embedded programming evaluated for Lusus there is no underlying operating system. The Lusus program would have to control the entire device. There were functions provided to interface with various parts of the device (i.e. serial ports) but there was no operating system to manage memory or schedule threads or control interrupts and synchronization.

The decision to use regular PCs allowed a far more rapid development cycle than using microcontrollers. As part of this bugs and protocol issues could more easily be identified and corrected. This process of platform evaluation and embedded development

experiments yielded interesting results for Lusus. For instance the process emphasized the limitations of embedded platforms. Having a very good idea of possible target hardware and software systems helps to keep Lusus simple and lightweight. The entire implementation, shown in Appendix B, is under 1600 lines of code, including 683 lines of code for SLIP and for handling the serial port under Linux, and 55 lines for computing the CRC. The compiled (and stripped) object files for Linux/x86 fit in 18 KB.

3.9 Chapter Summary

This chapter has presented some information on the testing implementation of Lusus. This implementation has yielded information on what works and what does not work in Lusus. Fortunately for the designers the functioning parts far outweigh the non-functioning parts. The functioning portions include route formation, data transmission, and data combining. Furthermore the parts that did not work quite as expected have been fixed. The prime example of a non-functioning part is the CSMA portion which was fixed by introducing jitter before each (re)transmission. Overall the implementation has provided a demonstration that the protocol is effective in small networks, and motivated an interest in how the protocol might perform in larger networks.

Chapter 4

Protocol Evaluation

4.1 Introduction

Following the general discussion of the operation of Lusus and a small implementation, Lusus must be evaluated to determine how well it works. The testing implementation served well as a proof-of-concept as well as a platform from which to base an analysis. However, a network of four nodes does not provide much in terms of performance predictions for networks ten or a hundred times larger.

Simple protocols lend themselves quite well to analysis. Their simple nature helps when describing them mathematically. This mathematical description is of vital importance to any analysis. More complex protocols (GRAd, Directed Diffusion) are more easily simulated as they are more complex and would require significantly more work to analyze.

Analysis gives more general results than simulation. This is to say that the results from a proper analysis are applicable across a wide range of scenarios, while simulation results might be assumed valid over scenarios that were not simulated, but there is no guarantee that they will be so. The data from simulations are the result of the particular scenario used to generate them.

This chapter starts off with a listing of the assumptions made throughout the evaluation. Next, it lists the symbols used throughout this chapter and their respective meaning. The rest of the chapter examines Lusus under various performance metrics and presents applications of these results. These performance metrics include: efficiency, Normalized

Routing Load, network capacity, and scalability. Efficiency and Normalized Routing Load give some details of how many bytes must be transmitted for every unit of data. Network capacity provides some information on the maximum number of bytes a Lusus network can transmit per unit time. Finally, scalability goes into some depth about how many nodes a Lusus network can contain and yet still function.

4.2 Assumptions and Definitions

Collisions Except where stated otherwise, it is assumed that Lusus is able to perfectly schedule packet transmissions such that no collisions occur. The effect of collisions is considered in Section 4.8.

Capacity The capacity of a wireless network has been defied by Gupta and Kumar [13] as the average number of bits per second that can be transmitted by every node to a destination.

Node Placement Nodes are assumed to be placed in such a way as to optimize network capacity, as defined by Gupta and Kumar [13].

Transmission Range Each node is assumed to be configured with an optimal transmission range for maximum network capacity.

Overhead In this analysis Overhead is defined as the number of additional bytes needed to transmit an n-byte data field.

4.3 List of Symbols

S_h Size in bytes of a packet header.

S_a Size of an acknowledgment packet.

S_d Size of a datagram.

S_m Size in bytes of the metadata contained within a datagram.

S_u Size in bytes of the data unit contained within a datagram.

n Number of datagrams per packet.

O_d Size in bytes of the overhead per unit of data in a packet ($\frac{3S_h}{n} + S_m$).

B Rate of BSA (Base Station Announcement) transmission in BSAs per second.

N The number of nodes in a Lusus network.

W The transmission speed of a node radio in bits per second.

4.4 Effects of Data Combining

Without the use of data combining Lusus needs to send an entire packet header per datagram. Furthermore Lusus needs to send one packet per piece of data. For nodes with multiple sensors this requires several packets per sampling instance.

With respect to packet framing: while the packet framing algorithm used in Lusus (SLIIP) adds some overhead, this overhead is only a few percent of the size of the smallest packet so it is not considered here. See Section A.3.6 for more details.

Without using data combining, a node must send packet and datagram headers as well as wait for an acknowledgment packet for each piece of data. During this waiting period a node cannot transmit more information. So the overhead for a piece of data is:

$$O_d = S_h + S_m + S_a.$$

But since an acknowledgment packet is basically a packet header plus a single datagram one can write

$$S_a = S_h + S_d$$

Combining from above yields:

$$O_d = S_h + S_m + S_h + S_d.$$

Assuming that, as in the current version, a packet header is approximately equal to a datagram in size gives:

$$O_d = S_h + S_m + S_h + S_h = 3S_h + S_m$$

So the overhead associated with transmitting a single piece of data is equal to $3S_h + S_m$ bytes.

Using data combining, the packet header and acknowledgment overhead is amortized over the number of datagrams. The metadata overhead is proportional to the number of datagrams.

When using data combining, a node must transmit a packet header and n datagram headers as well as wait for an acknowledgment packet. However, this transmission of headers and acknowledgment packet is shared by the n datagrams transmitted in the packet. Note that a single acknowledgment packet can acknowledge n datagrams provided they are sent in a single packet. So the overhead for n datagrams is:

$$O_d = \frac{S_h + (S_m \cdot n) + S_a}{n}$$

As given above, an acknowledgment packet is basically a packet header plus a single datagram one can write

$$S_a = S_h + S_d.$$

Combining the two above statements yields:

$$O_d = \frac{S_h + (S_m \cdot n) + S_h + S_d}{n}$$

Assuming that, as in the current version, a packet header is approximately equal to a datagram in size gives:

$$O_d = \frac{S_h + (S_m \cdot n) + S_h + S_h}{n} = \frac{3S_h + (S_m \cdot n)}{n} = \frac{3S_h}{n} + S_m \tag{4.4.1}$$

So the use of data combining results in about $\frac{3S_h}{n} + S_m$ bytes of overhead for n pieces of data.

As an example, consider the current version of Lusus. In this version $S_h = S_d =$ 16 bytes and $S_m = 8$ bytes.

Without data combining the overhead is $3 \cdot 16 + 8$ or 56 bytes for a 8-byte piece of data.

With data combining the overhead is between 32 ($n = 2$) and 11 ($n = 16$) bytes per 8-byte piece of data (Figure 4.1).

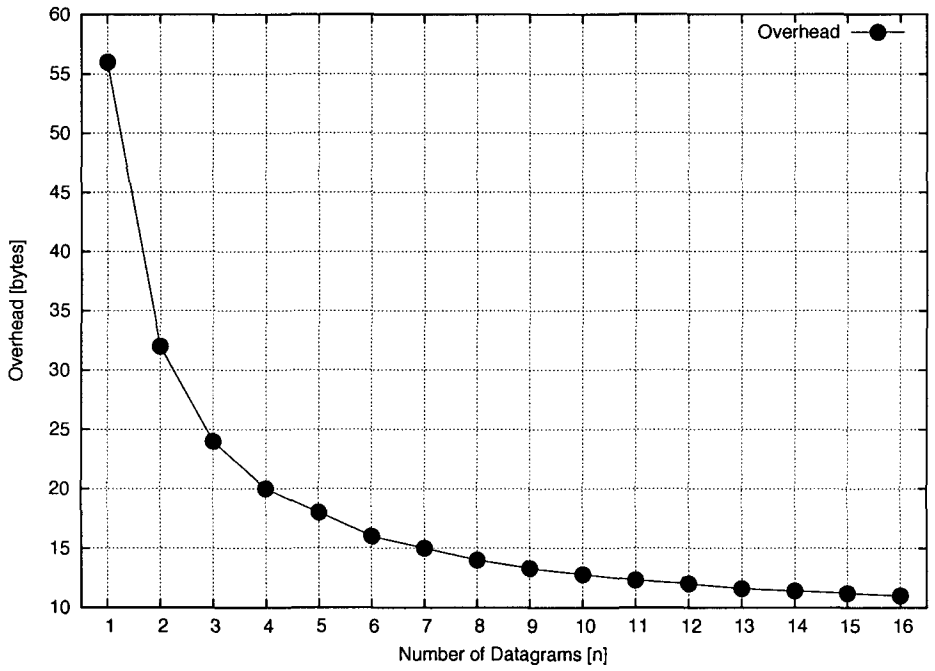

Figure 4.1: Bytes of Overhead per Datagram $\frac{3S_h}{n} + S_m$ vs Number of Datagrams n, with $S_h = 16,\ S_m = 8$.

It is important to note that the overhead as discussed in this section is for transmitting the full data field regardless of how much of it is actually used. This is due to the fact that Lusus uses a fixed-size data field, the entirety of which must be transmitted regardless of how much of it is actual data.

4.5 Effects of Data Rate on Data Combining

The previous section has shown that Lusus is more efficient when transmitting packets with more datagrams. However, this quest for efficiency should not be taken without consideration of the system as a whole. There are cases where a certain amount of inefficiency is acceptable.

A relatively small Lusus network (tens of nodes) transmitting a single bit once per day is unlikely to notice the impact of the greater overhead. In fact waiting until a few datagrams queue could negatively impact the data as some of it could take so long to arrive that it is less useful.

In a busy Lusus network (hundreds of nodes transmitting data several times per minute) waiting for a few datagrams to queue is a necessary step to achieving reasonable performance. Although it will delay the receipt of some pieces of data, the delay will help to significantly reduce the congestion, among other things, in the network.

Consider two nodes A and B. Both nodes sample data at the same rate. Node A sends data when it is collected. Node B waits several sampling periods before sending data in a single packet. Let n_a denote the number of datagrams per packet sent by node A and n_b denote the number of datagrams sent per packet by node B. Also Let P_a denote the number of packets transmitted by node A and P_b denote the number of packets transmitted by node B both during time t. $P_a > P_b$ because node B waits to collect several samples before transmitting. Given that both nodes send the same number of datagrams during time t, one can write:

$$P_a \cdot n_a = P_b \cdot n_b$$

Let O_a denote the overhead per datagram for each packet sent by node A and O_b denote the overhead per datagram for each packet sent by node B. From above the overhead per data item for transmitting n pieces of data in a single packet is $\frac{3S_h}{n} + S_m$. So

$$O_a = \frac{3S_h}{n_a} + S_m \text{ and } O_b = \frac{3S_h}{n_b} + S_m$$

This means that:

$$O_a > O_b \text{ since } n_a < n_b$$

The total amount of overhead for node A during time t is:

$$P_a \cdot O_a \cdot n_a$$

and for node B it is:

$$P_b \cdot O_b \cdot n_b$$

From this one can see that node A incurs more total overhead because

$$P_a \cdot n_a = P_b \cdot n_b \text{ and } O_a > O_b$$

This derivation shows how efficiency is increased by waiting for additional datagrams to accumulate before transmission in a single packet. The information presented and

derived in this section is useful for planning an implementation and deployment of Lusus. It will help determine the threshold for data packet transmission (i.e. number of datagrams that must accumulate before transmitting a data packet).

4.6 Normalized Routing Load

Normalized Routing Load (NRL) describes the amount of routing "overhead" in terms of data transmissions in a wireless network. Specifically NRL is defined as the ratio between number of routing packets transmitted and the number of successfully received data packets [25]. In this case each transmission of a routing packet across a single hop is counted as one transmission. However, each transmission of a data packet is not counted towards NRL; only the number of times a data packet reaches its respective destination.

For the calculation of NRL in Lusus recall that the ultimate destination of all data packets is the base station. Furthermore recall that the only routing packets Lusus uses are BSAs. By design each BSA is transmitted N times, where N is the number of nodes in the network. Assume that in a Lusus network each node samples S times per BSA. Also assume that there is no data combining or queuing; nodes transmit data as soon as it is sampled. This results in sending N BSAs for $S \cdot N$ data packets. Applying this to NRL yields

$$\frac{N}{S \cdot N} = \frac{1}{S} \text{ routing packet per data packet.} \tag{4.6.1}$$

For example consider a Lusus network that sends out a BSA once per hour and samples data every ten minutes. This means 6 samples per BSA, which results in an NRL of $\frac{1}{6}$ routing packet per data packet. Further discussion of NRL can be found in Section 5.7.

4.7 Capacity of Lusus

In a wireless ad-hoc network each node can transmit at most $O(\frac{W}{\sqrt{N \cdot logN}})$ bits per second [13]. This result holds for networks of N nodes which have been distributed over an area of one square meter. Each radio is assumed to transmit up to W bits/sec over a fixed range common to all radios in the network. Finally the network must schedule

transmissions in such a way that nodes do not interfere with each other. This is a very powerful result in that it allows the possible capacity of a wireless network to be determined in a relatively easy manner provided the assumptions behind it are valid.

There is no evidence to suggest that the performance of Lusus will significantly deviate from the result given above. It is entirely possible to create a scenario following the assumptions from [13] which produces a functional Lusus network.

Results suggest that the transport capacity of wireless ad-hoc networks improves when signal power decreases very rapidly over increasing distance [13]. This is to say that nodes are able to transmit more when the overall range of their transmissions is low and the network has many nodes. Furthermore there is hard evidence that wireless transmitters actually have two, usually different ranges: reception and interference [22]. Reception range refers to the maximum distance at which one node can successfully receive a transmission from another node. Interference range is the distance beyond reception range where a transmission can interfere with other node transmissions. In an ideal wireless network the two are the same, in practice the interference range usually exceeds the transmission range and may cause collisions that cannot be prevented by any existing protocol.

4.8 Scalability

The current version of Lusus does have several hard limits on scalability. From the protocol specification there is a limit of 16 hops between a node and its base station. This translates into a maximum network radius around a single base station of 16 hops. To help extend the size of a Lusus network multiple base stations can be used. When using multiple base stations each node in the network will automatically report its data to the nearest base station. This is accomplished by comparison of hop counts in BSA packets. If a node receives two BSA packets in series, the first with a hop count of ten and the second with a hop count of five, the node will transmit its data along the five-hop path. The node does not know (or care) if the BSAs came from the same or different base stations.

To determine the maximum density of Lusus, consider a network which does not transmit data packets. All this network is interested in doing is building routes. There must be sufficient transmission capacity to relay a BSA before the next one is transmitted.

Assume that the size of a packet header is approximately equal to the size of a datagram. So

$$\text{Size of a BSA} \approx 2S_h$$

Given this assumption each node needs to send approximately $2S_h$ bytes per BSA. As presented in Section 4.7 a network can transmit at most $O(\frac{W}{\sqrt{N \cdot logN}})$ bits per second. This most dense network needs to send

$$N \cdot 2S_h \text{ bytes per BSA}$$

So the maximum density is when the network is sending at the limit of capacity:

$$N \cdot 2S_h \cdot B \leq \frac{W}{\sqrt{N \cdot logN}}$$

In solving for N the equation will become:

$$N\sqrt{N}\sqrt{logN} \leq \frac{W}{2S_h \cdot B}$$

At this point one can substitute 2 for $\sqrt{logN}$ since for $177 \leq N \leq 575439$, $\sqrt{logN} \approx 2$ and the largest Lusus networks will be within that range. Substituting and solving for N yields:

$$N \leq \sqrt[3]{\left(\frac{W}{4S_h \cdot B}\right)^2} \tag{4.8.1}$$

As an example consider a deployment of a Lusus network. The nodes to be used will have wireless transceivers capable of 1 KBps (1000 Bytes per second). The nodes will use the current version of Lusus in which $S_h = S_d = 16$ bytes. Finally the base station for this network will send a BSA every five minutes.

So $W = 1000$ Bps; $S_h = 16$ bytes; $B = \frac{1}{300}$ BSAs per second. Using these values in the equation yields:

$$N \leq \sqrt[3]{\left(\frac{1000}{4 \cdot 16 \cdot \frac{1}{300}}\right)^2}$$

Reducing gives:

$$N \leq 280 \text{ nodes.}$$

As a further study of network density consider a Lusus network which transmits BSAs as well as data. The maximum density of this network must be less than the BSA-only network since the transmission of data requires a portion of the network's capacity. Also, one can use the previous result (Equation 4.8.1) to bootstrap a density with BSAs and data.

Let N_{BSA} denote the maximum density of nodes when transmitting only BSAs (Equation 4.8.1). Furthermore let D_b denote the amount of data transmitted per BSA. This includes the data generated by each node, times the number of retransmissions needed for the data to reach the base station. This means that the capacity of the network bandwidth used by data transmission is:

$$\frac{S_h}{S_h + D_b}$$

So factoring this into the equation for density yields:

$$N_{BSA} \cdot \frac{S_h}{S_h + D_b} \tag{4.8.2}$$

As an example consider the network given above. Assume that $D_b = 10$. Substituting values gives:

$$280 \cdot \frac{16}{16 + 10}$$

Reducing gives:

$$172 \text{ nodes.}$$

So in this example network with data transmission there can never be any more than 172 nodes.

For both examples of density one must remember that the figures given are for nodes all within an area of one square meter. This is to say that equations 4.8.1 and 4.8.2 need to be scaled for networks which occupy an area greater than one square meter. In [13] Gupta and Kumar recommend that capacity results be scaled by $\sqrt{A}$ where A is the area of the network in question. For example, if the network is deployed over one square kilometer, the maximum size of the network sending BSAs every 5 minutes and with $D_b =$ 10 becomes 172,000 nodes.

The most significant result obtained from this portion of the evaluation is that the density of a Lusus network in an area of one square meter grows at approximately

$O(B^{\frac{2}{3}})$ or $O(\sqrt[3]{B^2})$. This means that networks configured for frequent topology changes cannot achieve the high density that networks with infrequent topology changes have. This is because networks with frequent changes will require more BSAs to maintain connectivity than networks with infrequent topology changes.

4.9 Chapter Summary

This chapter has dissected the inner workings of Lusus in mathematical terms. It has provided some useful formulas for designers of Lusus networks. Furthermore it provides numerical results from which to analyze Lusus and draw conclusions about its effectiveness. The analysis includes formulas for the effects of data combining, Normalized Routing Load, network capacity and scalability.

Chapter 5

Discussion

5.1 Introduction

This chapter provides a discussion of the benefits of the Lusus MAC layer, tuning, cross-layer integration, and persistent routes. Furthermore, the chapter puts the evaluation results into context alongside comparable results from other protocols such as GRAd and flooding.

5.2 Lusus MAC

The MAC used in Lusus (a variant on CSMA) prevents certain types of collisions. However, there is a more complicated MAC which helps to prevent more types of collisions than CSMA. This MAC is known as CSMA/CA[1]. CSMA/CA uses small messages to "reserve" the communications medium for a set period of time[2]. This method of medium reservation helps to prevent collisions whereby a third node cannot hear the sender, but is within the range of the receiver and might interfere with the communication[3]. The small messages are known collectively as RTS/CTS[4] messages. A node wishing to transmit must first send an RTS packet and wait to receive a CTS packet from the intended recipient before transmitting the data packet.

[1]Carrier Sense Multiple Access with Collision Avoidance, as in IEEE 802.11.
[2]This reservation is usually for the amount of time it will take to transmit the packet.
[3]This is known as the Hidden Terminal Problem.
[4]Request To Send/ Clear To Send as in IEEE 802.11 [17].

The exchange of RTS/CTS packets incurs overhead in terms of time and network capacity [26]. The RTS/CTS exchange must occur before a regular packet can be transmitted. This delay is dependent on the speed of the transceivers and overall network traffic. Also, in ad-hoc networks this RTS/CTS exchange must take place at each hop along the path of a packet. This serves to increase the delay in reception of a packet. Furthermore, the RTS/CTS packets must be transmitted on the same medium used for data transmission. This reduces the capacity of the network to carry data packets.

Lusus can tolerate some packet loss due to collisions with a negligible impact on performance. To understand this consider two nodes A and B. Node A transmits a data packet with six datagrams to node B. Due to interference the packet is lost. In the mean time node A samples some additional data. Using data combining node A adds the new data to the original packet and transmits it. This process actually increases the efficiency due to an increase in the level of data combining. The downside is that there is an additional delay in the receipt of the data. Even without this additional data combining, the downside is really just a larger delay.

Retransmission does help to compensate for packets lost due to collision or corruption. Lusus has two forms of retransmission: explicit and implicit. Explicit retransmission is used for data packets. Implicit retransmission is a part of the periodic nature of BSAs. On a meta level the content of a BSA changes very little over time. The specific values contained within a BSA (the distance from the base station) do change, but even these do not change much. As such at the node level each BSA is in some sense interchangeable. So the periodic transmission of these BSAs helps to compensate for BSAs that are lost along the way. The only disadvantage is for networks with a changing topology. It is very possible for these networks to have temporary connectivity loss due to topology changes and lost BSAs.

5.3 Route Tuning

Lusus can be tuned based upon the anticipated mobility of the network. This tuning has an effect on the amount of data the network can carry. For example, in low mobility situations parameters can be adjusted to allow more data transmissions. This is

done by reducing the frequency of BSA's and also reducing the frequency of route table flushing at each node. Note that this is a low-mobility situation where the nodes are awake most of the time.

For networks with high mobility, the same parameters can be adjusted to help maintain connectivity. The tradeoff is the bandwidth use for BSAs decreases the amount of capacity available for data transmissions.

5.4 Higher Level Protocols

In most ad-hoc wireless networks, a higher-level protocol[5] is required in order to make use of the underlying wireless protocol. The most popular higher-level protocol is IP [10] [6]. Given the extensive use of IP and higher-level protocols (TCP [11] and UDP [9]) in other networks it does make sense to run ad-hoc wireless networks with these protocols. However, IP, TCP, and UDP were all designed for different systems and under different assumptions. For example IP was originally designed to provide connectivity between computers over wired connections. Furthermore packet loss and link interruption degrade the performance of TCP over wireless connections [15].

Lusus was designed to be a complete solution for wireless sensor networks. As such it is able to operate a network of sensor nodes without the need for additional protocols or software. In a simple computer there need not be a traditional operating system. The function of these simple computers is to gather sensor data via analog-to-digital converters and transmit this information to the users. These functions are easily accomplished by a single program. This single program can easily be an implementation of Lusus, augmented with the code needed to collect sensor data and to manage the sensor node.

As a direct result of Lusus being a complete solution, the hardware and software necessary to support a Lusus network need not be complex or expensive. Initial indications suggest that the parts for a complete Lusus node could cost as little as $20. A complete, well-documented software implementation of Lusus could be done in about 1000 lines of C. For what it does, this is a very small code base. The relatively small size of the source code makes implementation and debugging much simpler.

[5] A higher-level protocol is one that uses the features of another, lower-level protocol to communicate.

5.5 Lusus and Wireless Sensor Networks

The primary aim of wireless sensor networks is to gather sensor data from distinct points in the field [2]. This data is then collected in a central location for analysis. In this paradigm the vast majority of information is traveling towards a central point.

Lusus is optimized for this mostly one way flow of information. Most other protocols [23] [19] [10] are designed for an any-to-any transmission scheme. In this scheme any node must be allowed to send data to any other node in the network. Supporting this feature requires network capacity and resources on each node. This feature is not used very often, if at all, in wireless sensor networks. This simple design assumption in Lusus allows for an increase in efficiency and a corresponding reduction in complexity.

5.6 Persistent and Specific Routes

Persistent routes are helpful in that a series of packets can be transmitted without the need for route discovery or creation. The process of route discovery significantly increases the normalized routing load for a network (Section 4.6). Protocols such as GRAd use a method of route discovery as the core of their respective routing algorithms.

When flooding, each node must retransmit each and every packet that arrives. For a network of N nodes using flooding each packet must be transmitted at least N times. Assuming a constant transmission speed, as N increases the capacity of the network as a whole decreases. Furthermore even if the destination of a packet is a neighboring node, the network as a whole must still retransmit the packet.

GRAd is a bit more efficient than Flooding in terms of packet retransmission. However, it still allows a packet to be retransmitted many times more often than Lusus. This is because GRAd uses as many neighbors as possible to relay a packet to the destination. This method is basically a controlled form of flooding. In essence GRAd relays packets by using localized floods.

Each node in a Lusus network uses a specific next hop node to transmit the majority of its packets using a single node to relay packets leads to significantly less message replication, and in turn less network capacity is used to deliver a message. Assuming the

length of a route is L hops Lusus will transmit the packet L times. GRAd will transmit the packet $L + x$ times where x is the number of additional hops used by duplicate packets traveling from different neighbors, and flooding will transmit the packet N times (where $L + x \leq N$).

5.7 Normalized Routing Load

When comparing to other protocols, Lusus has a very favorable NRL. In comparison GRAd has an NRL of between 6 and 8 [23]. This means that for each packet received at its destination, there were approximately 7 additional routing packets transmitted. As stated in [23] GRAd counts every packet transmitted towards NRL. This is because each packet serves as a routing packet because it conveys information used in the cost table. In a comparable Lusus network the NRL would be less than one. In protocols like GRAd NRL is usually a function of network size and topology. This is because route lengths increase and the number of packets used to traverse these routes must increase as well. In Lusus NRL is only dependent on the ratio of BSAs to average number of data packets transmitted per node. Lusus only has a high NRL when it sends more BSAs than data packets. This is a possible, yet very unlikely condition. Even if this condition occurs, it is most likely the result of a network design decision. In such a case maintaining connectivity would be more important than a good NRL value.

NRL is a good measure of routing "overhead". However, there are ways to achieve a lower NRL while decreasing performance. Consider the results from Section 4.6. There are two ways to reduce NRL: send fewer BSAs or send more data packets. Sending fewer BSAs is possible, however as Section 5.3 points out there is a limit to this practice. The other method, sending more data packets, is of false economy. First and foremost, sending more data packets increases network load. Also, from Section 4.4 the amount of overhead increases as a Lusus network sends a greater number of smaller packets.

5.8 Data Combining

As presented in Section 4.4 Lusus increases its efficiency by combining data from several incoming packets into one outgoing packet. This modularity also allows Lusus to recover from lost packets in a graceful manner (Section 5.2). In other protocols such as GRAd and Flooding one packet per sampling interval for each node in the network must be relayed to the ultimate destination of data. Thus the data destination receives one packet per node per sampling interval. Lusus will, under the right conditions, reduce the number of packets being relayed and thus reduce the number of packets received at the data destination (base station). This reduction of packets received does not reflect a reduction of data received.

If a wireless sensor network is likened to a bucket brigade, GRAd and Flooding require nodes to relay each bucket that passes through. Lusus allows nodes to combine the meaningful contents of several buckets into one bucket for relaying. In this way Lusus makes better use of the unique communication topology of wireless ad-hoc networks.

5.9 Chapter Summary

This chapter has presented a discussion of Lusus performance. It started by talking in depth about the Lusus MAC and how well certain parts of it work. The discussion progressed through route tuning and data combining; both concepts of key importance to Lusus. Some comparison was given between Lusus, flooding, and GRAd. With respect to Normalized Routing Load Lusus evaluates quite well in comparison to GRAd.

Chapter 6

Future Work

Full-scale implementation The largest Lusus network to date contained four nodes arranged in a logically straight line. In the future we would like to build a Lusus network of tens or hundreds of nodes monitoring a forest. These nodes can be constructed of microcontrollers and simple serial radios. Furthermore the nodes might have basic sensors for things such as light level, temperature, and humidity. As a part of a full-scale implementation, multiple base stations would most likely be required.

Field Deployment In addition to the full-scale implementation we would like to deploy a Lusus network as part of a field study. In order to do this the nodes will have to contain all the features of the full-scale implementation in addition to a few more. Field nodes must be self-sufficient in terms of power. Ideally the nodes will have a combination of battery and solar power. Also, the nodes must be packaged in such a way that the electronics will not be damaged by the environment (i.e. moisture).

More Intelligent Data Combining In a future version of Lusus nodes could more intelligently multiplex data to reduce network load. In the current version of Lusus datagrams contain a 64-bit field. This field supposedly contains the data from a single sensor. This data can be as small as a single bit. Future versions of Lusus could allow the data from multiple sensors to be packed into a single datagram. This would allow for the more efficient use of the data field. Furthermore it will allow smaller packets which will in turn use less of the available bandwidth.

Chapter 7

Conclusion

Lusus is a simple, lightweight protocol that is a full solution for wireless sensor networks. It is simple to describe, implement, and analyze. Lusus is lightweight because it can be implemented with relatively little code. It also does not require much in terms of computing or communication resources. Lusus can be used to control and collect data from a network of nodes configured to be a wireless sensor network.

Lusus introduces the concept of modular data representation at the protocol level in wireless sensor networks. This modularization of data provides flexibility for nodes with several units of data to send. In fact due to the relaying present in wireless sensor networks, it is likely that these data units originated from multiple different nodes.

Lusus has been proven functional by way of implementation. The implementation of Lusus was an important step forward in its development as a protocol. Things about Lusus that didn't work quite right were discovered as a result of this implementation, specifically the difficulty of implementing CSMA with some real radios and the problems with collisions. These things were also fixed in subsequent versions of the protocol.

As a counterpart to the implementation, Lusus was evaluated and analyzed. This was a necessary step as the implementation was done on a significantly smaller scale than the scale Lusus was designed for. The evaluation took the form of mathematical analysis. Mathematical formulas explain the behavior of Lusus under different conditions, especially extremes of network size and communication load.

As a result of the evaluation it can be said that Lusus is better than flooding and GRAd. For one Lusus can deliver packets to their destination with far less replication (and

thus more efficiency) than flooding. Also Lusus can at worse meet and at best far exceed the replication rates in GRAd. Lusus establishes routes in a far more efficient manner than GRAd. With data combining Lusus incurs very little overhead per data unit sent. Lusus can be used for networks with hundreds of nodes even when the radios have low data rates, e.g. 1,000 bytes per second.

In conclusion Lusus is a useful protocol for running wireless sensor networks where data is sent to one or more central base stations.

Appendix A

Low-level protocol details

A.1 Introduction

This section can be thought of as an implementer's guide for Lusus. It contains various low-level details required for an implementation, but not required for an understanding of the protocol. Values prefixed with `0x` are in hexadecimal.

A.2 Constants

- Base Station Announcements are sent with a hop count of 0.
- The epoch is 12:00:00 AM January 1st 2000.
- Current preamble: `0x10f0f0f0` (4 bytes), used for the current version of Lusus, version 1.

A.3 Packet Header

All multi-byte quantities are sent in big-endian byte order.

A.3.1 Packet Types

- `0x0010` Base Station Announcement (BSA)

- `0x0020` Data Packet (DP)
- `0x0040` Urgent Data (URG)
- `0x0080` Ping Packet (PP)
- `0x0100` Traceroute Packet (TP)
- `0x0200` Node Command Packet (NCP)
- `0x0400` Acknowledgment Packet (ACK)

A.3.2 Packet Header

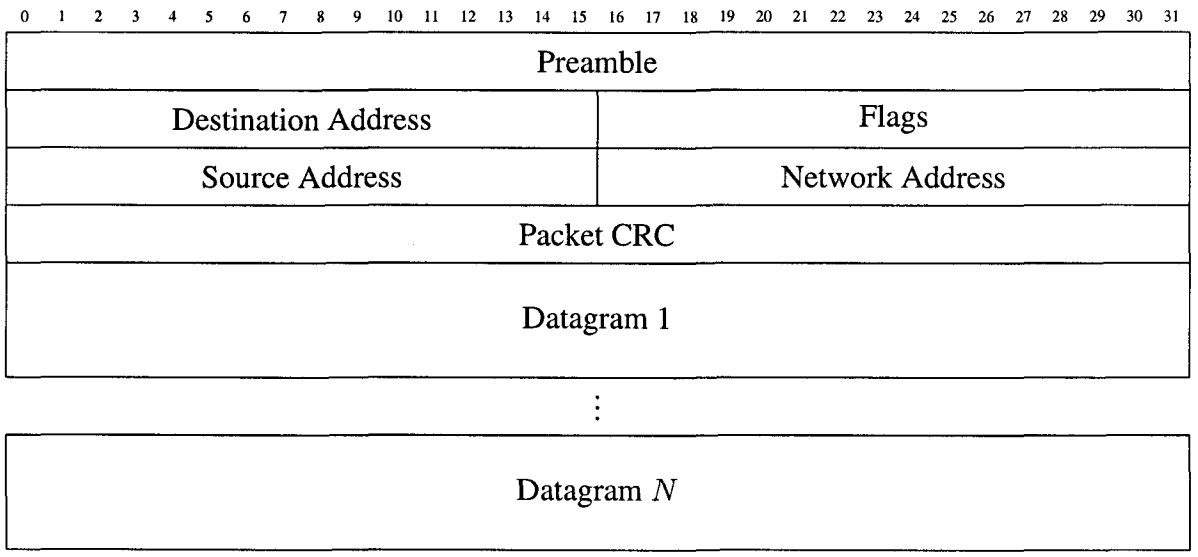

Figure A.1: The header of a Lusus packet

Preamble (32): Thirty-two bits used to indicate that what follows is a Lusus packet. A node is allowed to shut off its radio if the preamble does not match the specified preamble.

Destination Address (16 bits): The address of the intended recipient of a packet. The addresses `0x0000` and `0xffff` are reserved for system use. All other addresses are valid for nodes. The address `0xffff` is defined as the broadcast address. All systems receiving a packet addressed to `0xffff` are required to examine it for commands and information.

Source Address (16 bits): The address of the sender of a packet. The addresses of `0x0000` and `0xffff` are reserved for system use. All other addresses are valid for nodes.

Network Address (16 bits): The address of the virtual group that a set of nodes belongs to. The addresses `0x0000` and `0xffff` are reserved for system use. All other addresses are valid for networks.

Packet CRC (32 bits): A 32-bit CRC. This CRC covers the entire header plus all datagrams. When computing the CRC for a packet, this field must be initialized to zero.

Datagram: The datagram(s) contained within the packet. A single Lusus packet can have at most 16 datagrams.

A.3.3 Packet Flags

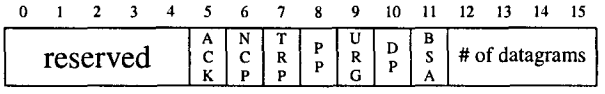

Figure A.2: The Flags field of a Lusus packet header

A.3.4 Datagram Header

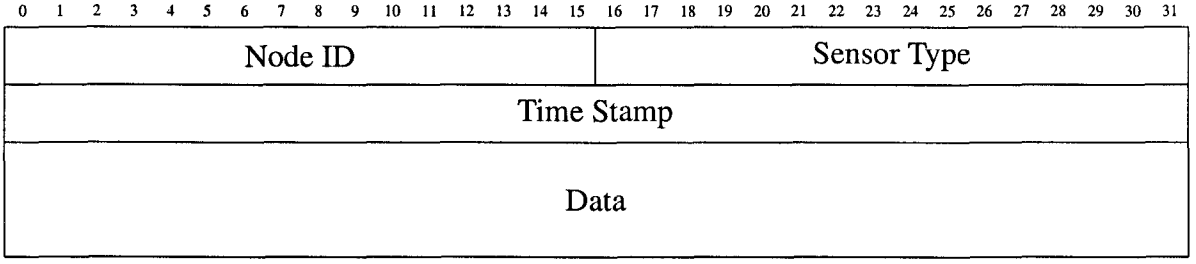

Figure A.3: A datagram header

Node ID (16): The sixteen-bit address of the node that collected this piece of data. Since this is similar to the source and destination addresses of the main packet header, the same restrictions apply here as they do there.

Time Stamp (32): Thirty-two bits used to denote the time at which this piece of data was collected. This field stores time as the number of seconds since the given epoch.

Data (64): A unit of data from the sensor type designated by the sensor type field. This piece of data can be at most sixty-four bits. If additional space is required, implementors may subdivide the sensor type into enough units to handle the entire sample.

Sensor Type (16): Sixteen bits used to denote which type of sensor was used to collect the data of this datagram. The types `0x0000` to `0x000f`, and `0xffff` are reserved for system use. All other numbers between `0x000f` and `0xffff` are valid types. The exact sensor used for each type is implementation-dependent with the following exceptions:

- `0x0000` is used to denote a hop-count in BSA, traceroute, and ping packets.
- `0x0001` is used to denote a hop along a traceroute path.
- `0x0002` is used to denote an acknowledgment datagram.

A.3.5 Acknowledgment

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
0x000a	0x0002
0x09f88a64	
0x00000000 0x21d3ef08	

Figure A.4: An example Acknowledgment datagram

The example datagram given in Figure A.4 shows an actual datagram from the testing implementation. The first sixteen bits indicate that node `0x000a` is acknowledging the receipt of a data packet. The next sixteen bits indicate that this is an acknowledgment datagram. This acknowledgment is being sent at `0x09f88a64` seconds past the Lusus epoch as indicated by the next thirty-two bits. The following four bytes are unused in acknowledgment datagrams. The last thirty-two bits of the packet show that data packet being acknowledged by this datagram had a CRC value of `0x21d3ef08`.

A.3.6 Packet Framing

Lusus uses a version of the Serial Line Internet Protocol (SLIP) to handle packet framing [24]. SLIP defines four different byte sequences to help with packet framing. These byte sequences are listed in Table A.1. To start sending a packet Lusus transmits a single copy of the END byte. The effect of this END byte is the flushing of the communications channel at the receiver(s). This helps reduce framing errors due to channel noise or interference. Next Lusus begins sending the packet in a byte-wise fashion. Before each packet byte is sent Lusus must make sure that it is not a reserved byte. If the byte to be transmitted is equal to the END byte Lusus transmits a special two-byte sequence of ESCEND. The ESC byte is another special byte sequence defined by SLIP which is used to ESCape a transmission. If the byte to be sent is equal to the ESC byte Lusus instead sends the special two-byte sequence of ESCESC. This process of sending two bytes to represent one is known as *byte stuffing*. Once the entire packet has been sent, Lusus sends a final END byte. On the receiving side, the node is responsible for interpreting the special byte sequences correctly and reinserting the proper resultant byte. For example, if a node receives a two-byte sequence of ESCESC (`0xdbdd`) the node must replace this with the intended value of ESC (`0xc0`). Note that the ESC and END bytes are never sent by themselves as a part of a packet. This would confuse receivers in that they are designed to look at the stream of bytes one by one.

This method of packet framing does introduce some additional overhead. Each packet is guaranteed to have at least two bytes of additional overhead due to framing. The overhead due to byte stuffing is dependent on the contents of the packet. It can be as low as

Mnemonic	Value in hexadecimal
END	`0xc0`
ESC	`0xdb`
ESCESC	`0xdbdd`
ESCEND	`0xdbdc`

Table A.1: SLIP Byte Sequences

zero or as high as the size of the packet. In packets with a uniformly random distribution of bytes the probability of byte stuffing on any one byte is $\frac{1}{128}$.

The average overhead due to byte stuffing is significantly less than one percent of the packet size. Even when combined with the given two bytes of overhead this overhead due to framing is still significantly less than the size of even the smallest Lusus header. For this reason framing overhead was not included in the calculations of overhead from Chapter 4.

A.3.7 Retransmission Thresholds

In contention-based networks[1] retransmission timing is of critical importance. Since nodes do not know what retransmission "schedule" their neighbors are on, some amount of randomness helps avoid repeated collisions due to retransmission synchronization.

In order to maximize bandwidth utilization Lusus nodes use the smallest reasonable packet (a packet header plus a single datagram) to compute their retransmission schedule. If Lusus instead used the largest possible packet there would be large periods where nodes would be waiting for unreasonably high amounts of time to retransmit.

Using the symbols from Chapter 4 the time to transmit the smallest packet is:

$$T = \frac{S_h + S_d}{W} \text{ seconds}$$

[1]Networks such as Ethernet where nodes must "contend" to transmit over a shared carrier.

Combine this with a random factor $y : \{0 \leq y < 1\}$ and the retransmission counter $x : \{0 \leq x < 10\}$ keeping in mind binary exponential back-off and the result is:

$$\text{wait time} = \frac{S_h + S_d}{W} \cdot (y \cdot 2^x) \text{ seconds} \tag{A.3.1}$$

Each time a packet needs to be transmitted a Lusus node must perform this computation and wait the resulting number of seconds. In this version of Lusus assuming a 1000byte/s radio transceiver, the maximum retransmission time is just over a minute.

A.3.8 CRC

The CRC used in Lusus is CRC-32 [16]. This CRC was chosen because it has been in use for Ethernet for many years with satisfactory results. The advantage of a table-driven implementation is that it requires very little CPU resources to compute, which is important for systems with limited computing power, which Lusus was designed for. Specifically this CRC is denoted by the polynomial:

$$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^{8} + x^{7} + x^{5} + x^{4} + x^{2} + x + 1$$

Appendix B

Source Code for the Test Implementation

B.1 Introduction

This particular implementation was written in C under Linux. Standard C functions were used wherever possible to maximize portability. The implementation is made up of two distinct levels: the lower SLIP communications layer, and the Lusus layer. An open-source software package[1] was modified to provide the SLIP functions. The slipnet package is a program that handles SLIP encoding and decoding as well as data transmission. Programmers wishing to use the slipnet package need to compile their code in with the slipnet source. The external code is written to use certain public functions provided by slipnet to transmit and receive data.

The Lusus layer is the Lusus-specific code developed for the sample implementation. This layer handles data collection, message relaying, and data acknowledgment. The source code is split into several files for modularity. The code was divided along two main lines: initialization and packet processing. All the initialization code along with the main function are contained in the lusus.c file. The packets.c file contains all the packet processing code. This includes packet generation as well as processing of a received packet.

[1]The software package is the ttynet/slipnet combination from `http://www2.ics.hawaii.edu/~esb/2005spring.ics651/notes1.html`. Courtesy Edo Biagioni.

Finally the lusus.h file contains all of the global variables as well as definitions and prototypes.

The CRC functionality is contained in the crc.c file. This is a simple table-driven implementation in which the table is created the first time the main crc function is accessed.

B.2 lusus.h

```
/*The main header file for the lusus protocol.
 *Lusus is a peer-to-peer communications and routing protocol for
 *wireless sensor networks.
 *
 *Please visit http://www.lusus.org for more information.
 *
 *Protocol and implementation (c) Dan Morton
    (dmorton@hawaii.edu) 2003.
 */
#include <pthread.h>
#include <stdio.h>
#include <time.h>

#define BCAST_ADDR 0xffff

/*The standard Lusus version 1 preamble.*/
#define PREAMBLE 0x10f0f0f0

/*Mask for the datagram count portion of the flags field.*/
#define DGM 0x000f

/*Shorthand for the flag bits in the message header*/
#define BSA 0x0010 //base station anouncement
#define DP 0x0020  //data packet
#define UD 0x0040  //urgent data
#define PP 0x0080  //ping packet
#define TR 0x0100  //traceroute packet
#define NCP 0x0200 //node command packet
#define ACK 0x0400 //ack packet

#define DATAQUEUE_SIZE 16

#define MAX_SLIP_SIZE 1006
```

```
/*Implementation of a Lusus version 1 packet header.*/
struct lususHeader {
  unsigned int preamble;
  unsigned short dst;
  unsigned short flags;
  unsigned short src;
  unsigned short netaddr;
  unsigned int CRC;
};


/*Implemenation of a Lusus version 1 datagram.
 *The datal and datah fields are used to acces the
 *higher and lower 32 bits of the data field.
 */
struct dgram {
  unsigned short nodeID;
  unsigned short sensorType;
  unsigned int timeStamp;
  unsigned int datah;
  unsigned int datal;
};

/*Generic structure for storing packets (not part of Lusus).*/
struct packet {
  int numBytes;
  char data[MAX_SLIP_SIZE];
};

/*functions*/
void sendACK(int tty, char * buf, int numBytes);

void *recvThread(int tty, char * buf, int numBytes);

void *sendThread(int tty);

void sendBSA(void);

int install_slip_data_handler (int, void (*) (int, char *, int));

int write_slip_data (int, char *, int);

void data_handler(int, char*, int);

void sample_send_data(void);
```

```
int sendData(void);

void sample_data(void);

/*globals in each file that includes lusus.h*/
extern unsigned short myaddr;
extern unsigned short mynet;
extern pthread_mutex_t serialLock;
extern int isBaseStation;
extern unsigned short hopCount;
extern unsigned short nextHop;

/*A structure to keep track of the number of communication errors
    by type.*/
struct errors{
  unsigned int preamble;
  unsigned int crc;
};

/*Implementation of the datagram queue.
 *This is used for storing datagrams between transmissions.
 *
 *The pthread mutex is used to lock the structure to protect
 *against concurency issues.
 *The length field is a counter to keep track of queue length.
 */
struct dataqueue{
  pthread_mutex_t lock;
  struct dgram queue[DATAQUEUE_SIZE];
  int length, head;
};
```

B.3 lusus.c

```
/*
 *Main function for the testing implementation
 *of the Lusus protocol. This implementation
 *never retransmits packets.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <termios.h>
#include "lusus.h"

struct dataqueue dataqueue;
struct errors errorCount;

int fileOutputFD;
int serialOutputFD;
unsigned short myaddr;
unsigned short mynet;
pthread_mutex_t serialLock;
int isBaseStation;
unsigned short hopCount;
unsigned short nextHop;

int main(int argc, char *argv[]){

  unsigned char *curDate;
  time_t curTime;
  int i, fd;

  struct termios tio,newtio;

  //read command-line arguments
  if(argc!=5) {
    printf("Error in command-line arguments\n");

    printf("Usage:\n");
    printf("lusus <tty> <output_file> <node_address> <-b|-c>\n");
    printf("where <tty> is the serial device to use.\n");
    printf("<output_file> is the file to output data to.\n");
    printf("<node_address> is the Lusus address of this
       node.\n");
    printf("<-b|-c> -b indicates a base station, -c indicates a
       client\n");
    exit(1);
  }

  //parse command-line args
  myaddr = strtol(argv[3],NULL,16);

  if(strncmp(argv[4],"-b",2)) {
    isBaseStation = 0;
  }
```

```
    else if(strncmp(argv[4],"-c",2)) {
      isBaseStation = 1;
    }
    else {
      printf("error with last command-line argument.\n");
      exit(1);
    }

    //open relevant files
    curTime = time(NULL);
    curDate = ctime(&curTime);
    fileOutputFD = open(argv[2],O_RDWR|O_APPEND);
    write(fileOutputFD, curDate, strlen(curDate));

    //open serial port.
    fd = serialOutputFD = open(argv[1], O_RDWR|O_NOCTTY);
    bzero(&newtio, sizeof(newtio));
    cfmakeraw(&newtio);
    newtio.c_cflag |= B9600 | CRTSCTS | CS8 | CLOCAL | CREAD;

    /* set input mode (non-canonical, no echo,...) */

    newtio.c_cc[VTIME]    = 0;   /* inter-character timer unused */
    newtio.c_cc[VMIN]     = 1;   /* blocking read until 5 chars
       received */

    tcflush(fd, TCIFLUSH);
    tcsetattr(fd,TCSANOW,&newtio);

    //install slip data handler function.
    install_slip_data_handler(serialOutputFD, data_handler);

  #ifdef DEBUG
    printf("%d is outputFD\n", serialOutputFD);
  #endif

    hopCount = -1;

    i=1;

    bzero(dataqueue.queue, sizeof(struct dgram) * 10);
    pthread_mutex_init(&dataqueue.lock, NULL);
    dataqueue.length = 0;
    dataqueue.head = 0;

```

```
    if(isBaseStation){
      //loop forever sleeping for 30 seconds before sending a BSA
      while(1) {
100
        sleep(30);
        sendBSA();
        if(i%5==0){
          i=0;
105     }
        i++;
      }
    }
    else {
110
      while(1) {

        sleep(10);

115     /*clear routing information every 120 seconds.*/
        if(i%12==0) {
#ifdef DEBUG
            printf("clearing routing information...\n");
#endif
120         hopCount = -1;
            i=1;
            continue;
        }

125     /*sample and send data every 20 seconds.*/
        if(i%2==0) {
            //sample_data
            sample_data();
            //send_data
130         sendData();
        }

        i++;
      }
135 }

    return 0;
  }

140 /*
```

```
 *A function to sample data, place it into a datagram,
 *and queue the datagram for transmission.
 */

void sample_data(void) {

  struct dgram *dgram;
  int tail;

  pthread_mutex_lock(&dataqueue.lock);

  if(dataqueue.length > 0 && dataqueue.length < DATAQUEUE_SIZE) {
    dgram = &dataqueue.queue[(dataqueue.head + dataqueue.length)
      % DATAQUEUE_SIZE];
  } else {
    dgram = &dataqueue.queue[dataqueue.head];
  }

  dgram->timeStamp = time(NULL);

  dgram->nodeID = myaddr;

  /*Because this is a testing implementation, the
   *following constant values are used to aid in
   *debugging. In a real implementation this would
   *gather data from a sensor and place it into
   *the relevant field.
   */
  dgram->sensorType = 10; /*timestamp*/
  dgram->datal = 0xabcdef;
  dgram->datah = myaddr;

  dataqueue.length++;

  pthread_mutex_unlock(&dataqueue.lock);

}
```

B.4 packets.c

```
/*
 *This file contains all packet processing functions for
 *the testing implementation of Lusus. The functions in this
 *file can perform a combination of packet processsing, creation,
```

```
 *or retransmission.
 */

#include "lusus.h"

extern struct dataqueue dataqueue;
extern struct errors errorCount;
extern int fileOutputFD;
extern int serialOutputFD;

/*function to create and send a base station anouncement*/
void sendBSA(void) {

  struct packet pkt;
  struct lususHeader *hdr = (struct lususHeader *)&pkt.data;
  struct dgram *dgram = (struct dgram *)(hdr + 1);

  bzero(&pkt, sizeof(struct packet));

  pkt.numBytes = sizeof(struct lususHeader) + (sizeof(struct
     dgram)*1);

  //set packet address and flags
  hdr->src = myaddr;
  hdr->dst = BCAST_ADDR;
  hdr->flags = BSA;
  hdr->flags++;
  hdr->preamble = PREAMBLE;

  dgram->datal = 0;
  dgram->datah = myaddr << 16;

  dgram->nodeID = myaddr;
  dgram->timeStamp = time(NULL);
  dgram->sensorType = 0x0001;

  hdr->CRC = crc(pkt.data, pkt.numBytes);

  //send packet
  write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);
}

/*A function to send the datagrams contained within the
 *datagram queue. Note that this function assumes that
 *there are less than 16 datagrams in the queue. If there
```

```
 *are more, this function must be called until it returns zero.
 *
 *The function takes no argument and returns the number of
 *datagrams transmitted.
 */
int sendData(void) {

  struct packet pkt;
  struct lususHeader *hdr = (struct lususHeader *)&pkt.data;
  struct dgram *dgram = (struct dgram *)(hdr + 1);
  int numDgrams = 0;
  bzero(&pkt, sizeof(struct packet));

  if(myaddr <= 0){
    printf("no route to base station, not sending data\n");
  }

  //get lock on the data queue.
  pthread_mutex_lock(&dataqueue.lock);

  //check to see if there is more than one dgram to send.
  if(dataqueue.length < rand()%8) {
    pthread_mutex_unlock(&dataqueue.lock);
    return 0;
  }

#ifdef DEBUG
  printf("number of datagrams to send: %d\n", dataqueue.length);
#endif

  //set address, etc for packet.
  hdr->dst = nextHop;
  hdr->src = myaddr;
  hdr->preamble = PREAMBLE;
  hdr->flags = DP;

  //loop through the list of dgrams adding them to a packet.

  while(dataqueue.length > 0){
    memcpy(dgram, &dataqueue.queue[dataqueue.head], sizeof(struct
        dgram));
    dgram++;
    dataqueue.length--;
    dataqueue.head++;
    dataqueue.head %= DATAQUEUE_SIZE;
```

```
        numDgrams++;

        hdr->flags++;

    }

    /*
    while(dataqueue.length != 0) {
        memcpy(dgram, &dataqueue.queue[dataqueue.length-1],
            sizeof(struct dgram));
        dgram++;
        dataqueue.length--;
        numDgrams++;

        //increment number of dgrams in pkt hdr.
        hdr->flags++;
    }
    */

    //set pkt len in structure
    pkt.numBytes = sizeof(struct lususHeader) + (sizeof(struct
        dgram) * numDgrams);

    pthread_mutex_unlock(&dataqueue.lock);

    //compute crc on packet.
    hdr->CRC = crc(pkt.data, pkt.numBytes);

    //send packet.
    write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);

    return numDgrams;
}

/*Function to create and send a ping request packet.
 *This function is only supposed to be used by Base Stations.
 */
void sendPP(void) {

    struct packet pkt;
    struct lususHeader *hdr = (struct lususHeader *)&pkt.data;
    struct dgram *dgram = (struct dgram *)(hdr + 1);

    bzero(&pkt, sizeof(struct packet));
```

```
    pkt.numBytes = sizeof(struct lususHeader) + sizeof(struct
        dgram);

    //set packet address and flags
    hdr->src = myaddr;
    hdr->dst = BCAST_ADDR;
    hdr->flags = PP;
    hdr->CRC = crc(pkt.data, pkt.numBytes);
    hdr->preamble = PREAMBLE;

    dgram->datal = 0;
    dgram->nodeID = myaddr;
    dgram->timeStamp = time(NULL);
    dgram->sensorType = 1;

    //send packet
    write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);

}

/*Function to create and send a traceroute packet.
 *This function is only supposed to be used by base stations.
 */
void sendTR(void) {

    struct packet pkt;
    struct lususHeader *hdr = (struct lususHeader *)&pkt.data;
    struct dgram *dgram = (struct dgram *)(hdr + 1);

    bzero(&pkt, sizeof(struct packet));

    pkt.numBytes = sizeof(struct lususHeader) + sizeof(struct
        dgram);

    //set packet address and flags
    hdr->src = myaddr;
    hdr->dst = BCAST_ADDR;
    hdr->flags = TR;
    hdr->flags++;
    hdr->preamble = PREAMBLE;

    dgram->datal = 0;
    dgram->nodeID = myaddr;
    dgram->timeStamp = time(NULL);
    dgram->sensorType = 0x0001;
```

```
    hdr->CRC = crc(pkt.data, pkt.numBytes);
180
    //send packet
    write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);

  }
185
  /*Callback function used by slipnet to deliver any incoming
     packets
   *to the Lusus layer of the testing implementation.
   */
  void data_handler(int tty, char * data, int numbytes) {
190
    struct packet pkt;
    struct packet tmp_pkt;
    struct lususHeader *hdr, *tmp_hdr;
    struct dgram *pdata, *tmp_pdata;
195
    unsigned int oldCRC;

    int numDgrams = 0;
    int lc = 0; //generic loop counter
200
    char outBuf[500];

    bzero(&pkt, sizeof(struct packet));
    bzero(&tmp_pkt, sizeof(struct packet));
205 bzero(&outBuf, 500);

  #ifdef DEBUG
    printf("lusus packets.c got a packet!\n");
  #endif
210
    memcpy(&pkt.data, data, numbytes);
    pkt.numBytes = numbytes;

    hdr = (struct lususHeader *)&pkt.data;
215 pdata = (struct dgram *)(hdr + 1);

    tmp_hdr = (struct lususHeader *)&tmp_pkt.data;
    tmp_pdata = (struct dgram *)(tmp_hdr + 1);

220 /*see if packet has proper preamble*/
    if(hdr->preamble != PREAMBLE) {
```

```
        errorCount.preamble++;
    #ifdef DEBUG
        printf("number of bad preambles: %d\n", errorCount.crc);
    #endif
        return;
      }

      /*check packet CRC*/
      oldCRC = hdr->CRC;
      hdr->CRC = 0;
      if(crc(&pkt.data, pkt.numBytes) != oldCRC){
        errorCount.crc++;

    #ifdef DEBUG
        printf("number of bad crc's: %d\n", errorCount.crc);
    #endif

        return;
      }

      if(hdr->src == myaddr) {
        //hearing my own packet, return
        return;
      }

      numDgrams = hdr->flags & 0x000f;

    #ifdef DEBUG
      printf("\tpkt has %d datagrams\n", numDgrams);
    #endif

      /*check packet dest addr*/
      if(hdr->dst == BCAST_ADDR) {

        if((hdr->flags & BSA) && !isBaseStation) {
          //base station announcement

    #ifdef DEBUG
          printf("node %x got a BSA from node %x with base station
             node %d which is %x hops away\n", myaddr, hdr->src,
             pdata->nodeID, pdata->datal);
    #endif

          if(pdata->datal < hopCount) {
            hopCount = pdata->datal;
```

```
265         pdata->datal++;
            nextHop = hdr->src;
            hdr->src = myaddr;

    #ifdef DEBUG
270         printf("node %x is %d hops from the base station and
               talks though node %x\n", myaddr, hopCount, nextHop);
    #endif

    #ifdef DEBUG
            /*Send my address in the unused portion of the
               datagram.*/
275         pdata->datah = myaddr << 16;
    #endif

            hdr->CRC = crc(pkt.data, pkt.numBytes);

280         //retransmit packet
            write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);
            return;
          }
          else {
285         /*Hopcount is greater than or equal to my current value.
             *Ignore this BSA and return.
             */
            return;
          }
290     }
        else if(hdr->flags & PP) {

    #ifdef DEBUG
          printf("node %d got a ping req from %d\n", myaddr,
             hdr->src);
295 #endif /*not implemented*/
          return;

        }
        else if((hdr->flags & TR) && !isBaseStation) {
300
    #ifdef DEBUG
    printf("node %x got a tracert req from %x with %x HC and mine is
       %x\n", myaddr, hdr->src, pdata->datal, hopCount);
    #endif

305       if(hdr->src == nextHop && pdata->datal == hopCount){
```

```
            //forward the packet along.

            hdr->src = myaddr;
            pdata->datal++;
310
            hdr->CRC = crc(pkt.data, pkt.numBytes);

            write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);

315         //now respond to the traceroute request.
            tmp_hdr->preamble = PREAMBLE;
            tmp_hdr->src = myaddr;
            tmp_hdr->dst = nextHop;
            tmp_hdr->flags = TR | 1;
320         tmp_pdata->nodeID = myaddr;
            tmp_pdata->datal = hopCount;
            tmp_pdata->sensorType = 1;
            tmp_pdata->timeStamp = time(NULL);

325         tmp_pkt.numBytes = sizeof(struct dgram) + sizeof(struct
               lususHeader);

            tmp_hdr->CRC = crc(tmp_pkt.data, tmp_pkt.numBytes);

            write_slip_data(serialOutputFD, tmp_pkt.data,
               tmp_pkt.numBytes);
330
            return;
          }
          else{
            //free the packet
335         return;
          }
        }

      }
340   else if(hdr->dst == myaddr) {
        //packet addressed to me

    #ifdef DEBUG
        printf("got a unicast pkt\n");
345 #endif

        if(hdr->flags & DP) {
          //data packet
```

```
        if(!isBaseStation){
350
#ifdef DEBUG
          printf("node %d got a data packet from %d. sending to
             %d\n", myaddr, hdr->src, nextHop);
#endif

355       //readdress packet
          hdr->dst = nextHop;
          hdr->src = myaddr;
          //tmp_pdata = 0;

360       //add any necessary data (data combining)
          pthread_mutex_lock(&dataqueue.lock);
          if(dataqueue.length > 0) { /*add data*/
            pdata += (hdr->flags & 0x000f);
            while((dataqueue.length > 0) && ((hdr->flags & 0x000f)
               < 16)) {
365           //copy data into packet.

              memcpy(pdata, &dataqueue.queue[dataqueue.head],
                 sizeof(struct dgram));
              pdata++;
              dataqueue.length--;
370           dataqueue.head++;
              dataqueue.head %= DATAQUEUE_SIZE;

              hdr->flags++;

375         }
            /*
              increment pkt size.
            */
            pkt.numBytes = sizeof(struct lususHeader) +
               (sizeof(struct dgram)*(hdr->flags & 0x00f));
380
          }
          pthread_mutex_unlock(&dataqueue.lock);


385


          hdr->CRC = crc(pkt.data, pkt.numBytes);

          //send packet along
```

```
390          write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);
        }

        else {

395 #ifdef DEBUG
          printf("base station got data\n");
    #endif

          numDgrams = hdr->flags & 0x000f;
400       for(lc=0; lc<numDgrams; lc++) {
            bzero(outBuf, 500);
            sprintf(outBuf, "%d,%x,%d,%d,%x\n",
                    pdata[lc].timeStamp,
                    pdata[lc].nodeID,
405                 pdata[lc].sensorType,
                    pdata[lc].datah,
                    pdata[lc].datal);
    #ifdef DEBUG
            printf("just got some data:\n%s", outBuf);
410 #endif
            write(fileOutputFD, outBuf, strlen(outBuf));
          }
        }
      }
415   else if(hdr->flags & PP) {
        //ping reply packet
    #ifdef DEBUG
        printf("ping reply packet\n");
    #endif
420   }
      else if(hdr->flags & TR) {
        //traceroute reply packet
        if(!isBaseStation){

425 #ifdef DEBUG
          printf("node got a traceroute reply from %x\n",
             hdr->src);
    #endif

          hdr->src = myaddr;
430       hdr->dst = nextHop;
          hdr->flags++;

          //here should check for space for another dgram
```

```
            tmp_pdata = pdata + numDgrams;
435         tmp_pdata->nodeID = myaddr;
            tmp_pdata->sensorType = 10;
            tmp_pdata->timeStamp = time(NULL);
            tmp_pdata->datal = hopCount;

440         pkt.numBytes += sizeof(struct dgram);
            hdr->CRC = crc(pkt.data, pkt.numBytes);

            write_slip_data(serialOutputFD, pkt.data, pkt.numBytes);

445         return;
          }
          else {
            //base station receiving a traceroute reply.

450 #ifdef DEBUG
            printf("base station got a traceroute reply.\n");
    #endif
            numDgrams = hdr->flags & 0x000f;
            for(lc=0; lc<numDgrams; lc++) {
455           bzero(outBuf, 500);
              sprintf(outBuf, "%d,%x,%d,%d,%d\n",
                      pdata[lc].timeStamp,
                      pdata[lc].nodeID,
                      pdata[lc].sensorType,
460                   pdata[lc].datah,
                      pdata[lc].datal);
    #ifdef DEBUG
              printf("just got some data:\n%s", outBuf);
    #endif
465           write(fileOutputFD, outBuf, strlen(outBuf));
            }

            return;
          }
470     }
      }
      else {
        //packet not for me, return.

475 #ifdef DEBUG
        printf("got %x's packet.\n", hdr->dst);
    #endif
```

```
        return;
480   }

    }
```

B.5 slipnet.c

```
 0 /* slipnet.c: provide serial-line send and receive of packets */
   /* link with simnet and pthreads */
   /* released under the GPL */

   #include <stdio.h>
 5 #include <pthread.h>

   #include <stdio.h>
   #include <sys/types.h>
   #include <sys/stat.h>
10 #include <fcntl.h>

   #define MAX_SLIP_SIZE    1024
   #define MAX_SLIP_SEND    1006
   #define END              0300    /* indicates end of packet */
15 #define ESC              0333    /* indicates byte stuffing */
   #define ESC_END          0334    /* ESC ESC_END means END data
      byte */
   #define ESC_ESC          0335    /* ESC ESC_ESC means ESC data
      byte */

   #define MAX_TTYS         100
20
   /* exported functions, could be in a .h file */
   int install_slip_data_handler (int, void (*) (int, char *, int));
   int write_slip_data (int, char *, int);
   void proc_fn(void *);
25
   /* buffers for the data */
   static char receive_buffer [MAX_TTYS] [MAX_SLIP_SIZE];
   /* we will use a negative receive_position to indicate that the
      last
    * character was an escape character */
30 static int receive_position [MAX_TTYS];
   /* true if an error was detected in the current frame */
   static int error_frame [MAX_TTYS];
   /* serialize all access to the buffers */
```

```
static pthread_mutex_t receive_mutex [MAX_TTYS];
pthread_mutex_t send_mutex [MAX_TTYS];
/* serialize access to the global data */
static pthread_mutex_t global_mutex = PTHREAD_MUTEX_INITIALIZER;
/* the data handlers are also global. */
typedef void (* my_data_handler) (int, char *, int);
static my_data_handler slip_data_handler [MAX_TTYS];

struct proc_info {
  unsigned char buf[MAX_SLIP_SIZE];
  int tty;
  int size;
  pthread_mutex_t lock;
} proc_info;

static void print_packet (char * string, char * data, int
    numbytes)
{
  int i;

  printf ("%s:\n", string);
  printf("pkt is %d bytes.\n", numbytes);
  for (i = 0; i < numbytes; i++) {
    /* must mask the byte with 0xff, since otherwise bytes
       greater
       than 0x80 will be converted to negative integers */
    printf ("%02x", (data [i]) & 0xff);
    if ((i == (numbytes - 1)) || (i % 20 == 19)) {
      printf ("\n");
    } else {
      printf (".");
    }
  }
}

static void put_char_in_buffer (int tty, unsigned char c)
{
  if (receive_position [tty] < MAX_SLIP_SIZE - 1) {
    receive_buffer [tty] [(receive_position [tty])++] = c;
  } else {
    printf ("error: slip framing error on port %d, maybe lost
       END\n", tty);
    /* discard the character — basically, we don't save it
       anywhere. */
    /* also make sure the current frame is discarded */
```

```
75      error_frame [tty] = 1;
      }
    }

    static void data_handler_for_tty (int tty, unsigned char c)
80  {
      pthread_t proc_thread;
    #ifdef DEBUGM
      printf ("received character %x/%o on port %d", c, c, tty);
    #endif /* DEBUGM */
85
      /* make sure we have been initialized */
      pthread_mutex_lock (&global_mutex);
      /* we have been initialized, so proceed */
      pthread_mutex_unlock (&global_mutex);
90    /* acquire the lock for the receive buffer */
      pthread_mutex_lock (&(receive_mutex [tty]));

      pthread_mutex_trylock(&(send_mutex[tty]));

95    if (error_frame [tty]) {
        if (c == END) {
          error_frame [tty] = 0;
          receive_position [tty] = 0;
          //unlock send mutex
100       pthread_mutex_unlock(&(send_mutex[tty]));
        }
      } else {
        if (receive_position [tty] < 0) {        /* last character
           was an escape */
          receive_position [tty] = - receive_position [tty];
105       if (c == ESC_END) {
            put_char_in_buffer (tty, END);
          } else if (c == ESC_ESC) {
            put_char_in_buffer (tty, ESC);
          } else {   /* this may be a legitimate oversight in the
             sender */
110         printf ("warning: accepting illegal character after
               ESC\n");
            put_char_in_buffer (tty, c);
          }
        } else {                          /* last character was not ESC
            */
          if (c == END) {                  /* done, give packet to data
              handler. */
```

```
115         if (receive_position [tty] > 0) {  /* packet is not empty
                */
            if (slip_data_handler [tty] == NULL) {
                printf ("error: received packet, but no slip data
                    handler\n");
                print_packet ("received packet", receive_buffer
                    [tty],
                                receive_position [tty]);
120             receive_position [tty] = 0;
            } else {
                //unlock send mutex so that the data_handler call can
                    send data.
                //pthread_mutex_unlock(&(send_mutex[tty]));
#ifdef DEBUG
125             /*print_packet ("received packet", receive_buffer
                    [tty],
                                receive_position [tty]);
                */
#endif /* DEBUG */
                pthread_mutex_unlock(&(send_mutex[tty]));
130             /* note the receive buffer remains locked while we
                    call the
                    slip data handler.  If the slip data handler never
                        returns,
                    slip will deadlock, i.e., be unable to ever again
                        receive data.
                    This would also block the receive thread in
                        ttynet. */

135 #ifdef DEBUG
                printf("trying to get a lock for the proc_buf\n");
#endif
                pthread_mutex_lock(&proc_info.lock);
                memcpy(&proc_info.buf, receive_buffer[tty],
                    receive_position[tty]);
140             proc_info.size = receive_position[tty];
                proc_info.tty = tty;
                pthread_create(&proc_thread, NULL, (void*)&proc_fn,
                    (void*)&proc_info);

            }
145         /* get ready to start receiving a new packet */
            receive_position [tty] = 0;
        } /* else: silently ignore packets of size 0 */
```

```
        } else if (c == ESC) {          /* signal for the next
            character */
          receive_position [tty] = - receive_position [tty];
150     } else {                          /* 'normal' character */
          put_char_in_buffer (tty, c);
        }
      }
    }
155 /* finally make the buffer available to other threads. */
    pthread_mutex_unlock (&(receive_mutex [tty]));
  }

  /* returns the identifier (an integer >= 0) to be used for
     write_slip_data */
160 int install_slip_data_handler (int tty,
                                  void (* data_handler) (int, char
                                     *, int))
  {
    int fd;
    pthread_mutex_t tmp = PTHREAD_MUTEX_INITIALIZER;
165
    /* keep thread from executing until we are done initializing */
    pthread_mutex_lock (&(global_mutex));
    fd = tty;
    install_tty_data_handler (tty, data_handler_for_tty);
170 if (fd < 0) {
      pthread_mutex_unlock (&global_mutex);
      return fd;
    }
    receive_position [fd] = 0;
175 error_frame [fd] = 0;
    memcpy (&(receive_mutex [fd]), &tmp, sizeof (tmp));
    memcpy (&(send_mutex [fd]), &tmp, sizeof (tmp));
    slip_data_handler [fd] = data_handler;
    pthread_mutex_unlock (&global_mutex);
180
    proc_info.size = 0;
    proc_info.tty = 0;
    pthread_mutex_init(&proc_info.lock, NULL);
    bzero(&proc_info.buf, MAX_SLIP_SIZE);
185
    return fd;
  }
```

```
/* this is a macro so the return statement returns from
   write_slip_data */
#define WRITE_BYTE(fd, c)                                  \
    if (write_tty_data (fd, c) != 1) {                     \
      pthread_mutex_unlock (&(send_mutex [fd]));           \
      printf ("slip: error writing tty data\n");           \
      return -1;                                           \
    }

int write_slip_data (int fd, char * data, int numbytes)
{
  int byte;
  unsigned long sleepval;
  fd_set wset;
  unsigned char c;

  int retval;
  struct timeval wait_time;
  struct timespec wait_time2;
  wait_time.tv_sec = 2;
  wait_time2.tv_sec = 0;
  wait_time2.tv_nsec = 1000000000 / (7200 / 8); // 8 bits at 9600
      b/s

  if ((numbytes <= 0) || (numbytes > MAX_SLIP_SEND)) {
    printf ("slip: bad size %d\n", numbytes);
    return -1;
  }

  //random jitter to avoid collisions.
  sleepval = random();

  //make sure we're gonna sleep for less than 1 sec.
  sleepval = sleepval % 10000000;

  usleep(sleepval);

#ifdef DEBUG
  printf ("acquiring send lock for tty %d\n", fd);
#endif /* DEBUG */
  pthread_mutex_lock (&(send_mutex [fd]));

  FD_ZERO(&wset);
  FD_SET(fd, &wset);
  retval = select(fd+1, NULL, &wset, NULL, &wait_time);
```

```
#ifdef DEBUG
  print_packet ("sending packet", data, numbytes);
#endif /* DEBUG */
  WRITE_BYTE (fd, END);

  for (byte = 0; byte < numbytes; byte++) {
    nanosleep (&wait_time2, NULL);

    c = (data [byte]) & 0xff;

    if (c == END) {
      WRITE_BYTE (fd, ESC);
      WRITE_BYTE (fd, ESC_END);
    } else if (c == ESC) {
      WRITE_BYTE (fd, ESC);
      WRITE_BYTE (fd, ESC_ESC);
    } else {                          /* normal byte */
      WRITE_BYTE (fd, c);
    }
  }
  WRITE_BYTE (fd, END);
  pthread_mutex_unlock (&(send_mutex [fd]));

  return numbytes;
}

void proc_fn(void * info) {

  struct proc_info *proc = (struct proc_info *)info;
  print_packet("received packet", proc->buf, proc->size);
  slip_data_handler [proc->tty] (proc->tty, proc->buf,
     proc->size);
  pthread_mutex_unlock(&proc->lock);
}
```

B.6 simnet.c

```
/* simnet.c: handle the details of receiving data from the serial
     tty.
   It is required by the slipnet.c file
   link with -lpthread
   released under the GPL
*/
```

```
5
  #include <stdio.h>
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <netinet/in.h>
10 #include <netdb.h>
  #include <fcntl.h>
  #include <termios.h>
  #include <unistd.h>
  #include <pthread.h>
15 #include <string.h>

  #define MAX_TTYS 25
  #define CONFIG_FILE "null"

20 /* this is the simulator configuration information we need */
  struct simulation_config {
    int socket;
    int in_use;
  };
25
  extern pthread_mutex_t send_mutex [MAX_TTYS];

  static struct simulation_config tty_sim [MAX_TTYS];

30 static int valid_ttys = 0;

  #define simerror(message) { perror (message); exit(1); }

  /* any static function is NOT exported */
35 static int initialize_tty (int tty_number)
  {

    printf("initting tty %d\n", tty_number);

40  if (tty_number >= MAX_TTYS) { simerror ("tty number"); }
    if (tty_sim [tty_number].in_use) { simerror ("tty already in
       use"); }
    tty_sim [valid_ttys].in_use = 1;
    return tty_number;
  }
45
  struct receive_thread_arg {
    void (* data_handler) (int, char);
    int tty;
```

```
};

static void * tty_receive_thread (void * argument)
{
  /* cast the argument back to a pointer to the
     receive_thread_arg */
  struct receive_thread_arg * rta = (struct receive_thread_arg
     *) argument;
  void (* data_handler) (int, char) = rta->data_handler;
  int tty = rta->tty;

  fd_set readfs;
  int select_val;
  struct timeval timeout;

  timeout.tv_sec = 1;
  timeout.tv_usec = 0;


  printf ("tty_receive_thread is starting\n");
  /* we have read the argument, it won't be used ever again, so
     free it */
  free (argument);
  /* set the argument to NULL to guarantee it won't ever be used
     again */
  argument = NULL;

  /* loop forever, and whenever data is received, call the data
     handler */
  /* when no data is available, the loop blocks on read. */

  while (1) {
    char buffer [1];

    int bytes;

    FD_ZERO(&readfs);
    FD_SET(tty, &readfs);
    //need select to timeout after a sec or two
    select_val = select(tty+1, &readfs, NULL, NULL, &timeout);

    if(select_val == 0) {
      pthread_mutex_unlock(&(send_mutex[tty]));
      continue;
    }
```

```
90      bytes = read(tty, buffer, 1);

        if (bytes == -1) {
          perror ("recvfrom");
          exit (1);
95      }
        if (bytes == 1) {
          /* deliver the data with an upcall */
          data_handler (tty, buffer [0]);
        } else {
100       printf ("ttynet error: got value %d from 'recvfrom',
             expected 1\n",
                  bytes);
        }
      }
      /* we never return, but if we ever did, we'd want to return a
         void *  */
105   printf ("error: returning from infinite loop\n");
      return NULL;
    }

    /* returns the identifier (an integer >= 0) to be used for
       write_tty_data */
110 int install_tty_data_handler (int tty, void (* data_handler)
       (int, char))
    {
      pthread_t thread;
      int actual_tty = initialize_tty (tty);
      struct receive_thread_arg * arg =
115     (struct receive_thread_arg *) malloc (sizeof (struct
           receive_thread_arg));

      printf("actual_tty is %d\n", actual_tty);

      if (actual_tty < 0) {
120     free (arg);
        return -1;
      }
      arg->tty = actual_tty;
      arg->data_handler = data_handler;
125   printf("creating thread for tty %d\n", actual_tty);
      if (pthread_create (&thread, NULL, &tty_receive_thread, (void
         *) arg) < 0) {
        perror ("pthread_create");
```

```
      exit (1);
    }
    return actual_tty;
  }

  int write_tty_data (int tty, int data)
  {
    unsigned char buffer[1];
    int retval;

    buffer[0] = data;

    retval = write(tty, buffer, 1);
    //printf("wrote char %x/%o on port %d\n", buffer[0], buffer[0],
        tty);

    return retval;

}
```

B.7 crc.c

```
/*This file implements the CRC functions for Lusus.*/
/*The code used here is taken from the PNG specification.*/
/*See http://www.libpng.org/pub/png/spec/1.2/PNG-CRCAppendix.html
    for more details.*/
#include "lusus.h"

/* Table of CRCs of all 8-bit messages. */
unsigned long crc_table[256];

/* Flag: has the table been computed? Initially false. */
int crc_table_computed = 0;

/* Make the table for a fast CRC. */
void make_crc_table(void)
{
  unsigned long c;
  int n, k;

  for (n = 0; n < 256; n++) {
    c = (unsigned long) n;
    for (k = 0; k < 8; k++) {
      if (c & 1)
```

```
            c = 0xedb88320L ^ (c >> 1);
          else
            c = c >> 1;
        }
25      crc_table[n] = c;
      }
      crc_table_computed = 1;
    }

30  /* Update a running CRC with the bytes buf[0..len-1]--the CRC
       should be initialized to all 1's, and the transmitted value
       is the 1's complement of the final running CRC (see the
       crc() routine below)). */

35  unsigned long update_crc(unsigned long crc, unsigned char *buf,
       int len)
    {
      unsigned long c = crc;
      int n;

40    if (!crc_table_computed)
        make_crc_table();
      for (n = 0; n < len; n++) {
        c = crc_table[(c ^ buf[n]) & 0xff] ^ (c >> 8);
      }
45    return c;
    }

    /* Return the CRC of the bytes buf[0..len-1]. */
    unsigned long crc(unsigned char *buf, int len)
50  {
      return update_crc(0xffffffffL, buf, len) ^ 0xffffffffL;
    }
```

Appendix C

Glossary

Upstream Towards the Base Station.

Downstream Away from the Base Station.

Node A computer system connected to a network.

Hop The portion of the path between source and destination nodes that lies between two adjacent nodes.

Base Station A central computer in a Lusus network.

CRC Cyclic Redundancy Check.

Unicast A packet addressed to a specific node.

Broadcast A packet addressed to all nodes.

Bibliography

[1] N. Abramson. The aloha system - another alternative for computer communications. In *Proceedings of Fall Joint Computer Conference*. AFIPS Conference, 1970.

[2] Edo Biagioni, K.W. Bridges, and Brian Chee. The PODS project: A Remote Ecological Micro-Sensor Network. `http://red2.ics.hawaii.edu/`.

[3] Edoardo Biagioni and Kent Bridges. The application of remote sensor technology to assist the recovery of rare and endangered species. *International Journal of High Performance Computing Applications*, 16(3):15, August 2002.

[4] Rachel Cardell-Oliver. Why flooding is unreliable in multi-hop, wireless networks. `http://www.cs.uwa.edu.au/research/Technical_Reports/UWA-CSSE-04-001/Report-04-001.pdf`, School of Computer Science and Software Engineering The University of Western Australia, 2004.

[5] Brian Chee. Personal communication, Circa 2002.

[6] Shu Hui Chen. Multipath on-demand routing in sensor network topologies. `http://red2.ics.hawaii.edu/~shuc/mor-thesis-2003.pdf`, University of Hawaii at Manoa, May 2003.

[7] D.J. Cook and S.K. DAS, editors. *Smart Environments: Technologies, Protocols, and Applications*. John Wiley, New York, 2004.

[8] DARPA. The Defence Advanced Research Projects Agency SENSIT project. `http://www.sainc.com/sensit/`.

[9] Jon Postel ed. RFC 768 - user datagram protocol. `http://www.ietf.org/rfc/rfc0768.txt`.

[10] Jon Postel ed. RFC 791 - internet protocol. `http://www.ietf.org/rfc/rfc0791.txt`.

[11] Jon Postel ed. RFC 793 - transmission control protocol. `http://www.ietf.org/rfc/rfc0793.txt`.

[12] Gerard Fryer. Personal communication, Circa 2002.

[13] P. Gupta and P. R. Kumar. The capacity of wireless networks. In *IEEE Transactions on Information Theory*, volume 46, Mar 2000.

[14] Hawaii International Conference on Systems Sciences. *Wireless Sensor Placement for Reliable and Efficient Data Collection*, Waikoloa, Hawaii, January 2003.

[15] Gavin Holland and Nitin Vaidya. Analysis of TCP performance over mobile ad hoc networks. *Wirel. Netw.*, 8(2/3):275–288, 2002.

[16] IEEE. *IEEE Standards for Local Area Networks: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.* IEEE, New York, New York, 1985.

[17] IEEE. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications: Higher Speed Physical Layer (PHY) Extension in the 2.4 Ghz Band.* IEEE, New York, New York, 1999.

[18] The University of Southern California Information Sciences Institute. The Network Simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[19] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[20] Jeong Geun Kim and Marwan M. Krunz. Bandwidth allocation in wireless networks with guaranteed packet-loss performance. *IEEE/ACM Trans. Netw.*, 8(3):337–349, 2000.

[21] S. Kurkowski, T. Camp, and M. Colagrosso. MANET simulation studies: The current state and new simulation tools. http://toilers.mines.edu/EXTERNAL/twiki/pub/Toilers/PublicationList/CSM-MCS-05-02.pdf MCS-05-02, The Colorado School of Mines, February 2005.

[22] Rohit Negi and Arjunan Rajeswaran. Physical layer effect on MAC performance in ad-hoc wireless networks. In *Proc. Commun.* Internet and Info. Tech., CIIT, 2003.

[23] Robert D. Poor. Gradient routing in ad hoc networks. http://www.media.mit.edu/pia/Research/ESP/texts/poorieeepaper.pdf.

[24] J. Romkey. RFC 1055 - nonstandard for transmission of IP datagrams over serial lines: SLIP. http://www.ietf.org/rfc/rfc1055.txt.

[25] E. Royer S. Das, C. Perkins. Performance comparison of two on-demand routing protocols for ad hoc networks. *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, Mar 2000.

[26] J. B. Carruthers S. Ray and D. Starobinski. RTS/CTS-induced congestion in ad hoc wireless LANs. *WCNC 2003: Wireless Communications and Networking Conference*, 2003.