NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

FILE TRANSFER WITH ERASURE CODING OVER WIRELESS SENSOR NETWORKS

by

Thomas Edward Childers

March 2009

Thesis Advisor:	John McEachen
Thesis Co-Advisor:	Murali Tummala

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE File Transfer with Erasure Coding over Wireless Sensor Networks	**5. FUNDING NUMBERS**
6. AUTHOR(S) Thomas Edward Childers	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

In order to provide a step towards the goal of passing TCP/IP traffic across wireless sensor networks, a method for file transfer utilizing forward error correction (FEC) is studied. Previous work in the area of terminal communication across the sensor network is expanded upon to include file transfer in order to provide a more capable channel and a basis for testing the performance obtained through erasure coding. The results of the FEC implementation are examined using multiple sensor network configurations. The study is completed with recommendations for continued work towards developing tunneled TCP/IP data transfer across wireless sensor networks.

14. SUBJECT TERMS Wireless Communication, Wireless Sensor Networks, Data Transmission	**15. NUMBER OF PAGES** 103
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

FILE TRANSFER WITH ERASURE CODING OVER WIRELESS SENSOR NETWORKS

Thomas Edward Childers
Lieutenant, United States Navy
Bachelor of Science in Electrical Engineering, Georgia Institute of Technology, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2009

Author: Thomas Edward Childers

Approved by: John C. McEachen
Thesis Advisor

Murali Tummala
Thesis Co-Advisor

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In order to provide a step towards the goal of passing TCP/IP traffic across wireless sensor networks, a method for file transfer utilizing forward error correction (FEC) is studied. Previous work in the area of terminal communication across the sensor network is expanded upon to include file transfer in order to provide a more capable channel and a basis for testing the performance obtained through erasure coding. The results of the FEC implementation are examined using multiple sensor network configurations. The study is completed with recommendations for continued work towards developing tunneled TCP/IP data transfer across wireless sensor networks.

.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ABBREVIATIONS AND ACRONYMS

ADC	Analog-to-Digital Converter
ARQ	Automated Repeat Request
CRC	Cyclic Redundancy Check
DARPA	Defense Advanced Research Projects Agency
DSN	Distributed Sensor Networks
FEC	Forward Error Correction
IDE	Integrated Development Environment
JPEG	Joint Photographic Expert Group
MAC	Medium Access Control
RAM	Random Access Memory
SNAIL	Sensor Network Adaptation Interface Layer
SNR	Signal-to-Noise Ratio
WSN	Wireless Sensor Network

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I am very thankful for the guidance and direction from Professor John McEachen, who allowed me to pursue the interesting topic of sensor networking and provided an ideal amount of experimental freedom during the research stages of this project.

The subject matter expertise obtained from Professor Murali Tummala greatly helped in the development of my research.

I would also like to thank Bob Broadston for his guidance and the use of the anechoic chamber which made the collection of more reliable experimental data possible.

Last but not least, I would like to extend my appreciation to all who have contributed to the completion of this thesis in one way or another.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Wireless sensor networks were originally developed to provide the warfighter with a more accurate picture of the battlefield environment using small, virtually undetectable devices. These devices, upon deployment, could form an ad hoc network, sense the physical conditions of their surroundings, and report back their observations without putting the life of a single soldier at risk.

While the sensor networks are optimized for this type of use, other applications may be equally beneficial to the warfighter. Utilizing existing systems to fit new mission requirements is one of the most cost and time effective acquisition strategies. Along this line, sensor networks may be able to provide a new service on the battlefield as a bridge for TCP/IP networks. This could provide a means for passing vital intelligence data and communications in areas with little or no infrastructure.

Based on this new idea, a former student of the Naval Postgraduate School devised a sensor network channel to transfer text messages from one terminal to another. Building upon this previous work, file transfer and forward error correction are now added to make a more capable and reliable channel. A form of block coding called erasure coding was implemented during this research. Several tests were conducted using different hardware configurations to examine the performance of the now enhanced channel.

While the file transfer tests were successful and the error correction method proved effective, channel weaknesses were observed. Larger network topologies appeared to present congestion issues that require further analysis. Also, observed throughput was well below the rates necessary for transfer of multimedia or other useful traffic. The Carrier Sense Multiple Access (CSMA) protocol used during the tests proved to be the limiting factor.

Ideas for future research include optimization of the erasure code used during the experiment, tests with a Time Division Multiple Access (TDMA) based medium access protocol, and tests with some form of feedback channel. These measures might be a positive next step towards the goal of passing TCP/IP traffic across a sensor network.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

Wireless networking continues to be one of the fastest growing technologies today. Through advances in technology, the cost and size of wireless devices have reduced dramatically, making them more readily available to people today than ever before. Whether it is a cell phone, wireless internet router, or some other device, the average person today is using wireless devices on an increasingly regular basis.

A small subset of wireless networking is wireless sensor networks. Much less known to the average consumer, the technology used by these networks is just now reaching the maturity level necessary to make them commercially viable. These sensor networks are made up of multiple low cost, spatially distributed, autonomous devices that are able to collect and distribute environmental information for various purposes. The devices are able to independently form an ad hoc network upon deployment and commence the mission they were intended for.

Military applications for sensor networks have been the driving force for much of the research being conducted today. Modern research in this area can be traced back to work done during the early 1980s by the Defense Advanced Research Projects Agency (DARPA), and their Distributed Sensor Networks program (DSN) [1]. Commercial applications have come along much more slowly but have begun to accelerate recently. Although the commercial benefits cannot be overlooked, the military aspect is especially intriguing due to the portability and small footprint of these wireless devices.

Sensor networks do not face some of the same limitations as normal networks in that they are battery operated and thus do not rely on an established infrastructure. They can be placed to operate in remote locations just as easily as they could in easier to access, more populated areas. As the technology progresses, the day will soon come when these devices are being dropped from the air into areas of interest. Upon landing, they will power up, establish communications with the other sensor nodes, and begin operations.

The advantages of these sensor devices cannot be denied, but they do have a few weaknesses that must be considered. While being portable, and free of the need for external power sources, the fact that they rely on batteries can also be a disadvantage. Typical sensor networks pass small amounts of data between nodes. These data exchanges require a small drain on the batteries. With this kind of implementation, the units will last for months on batteries. If the units are to provide a higher level of service, such as multimedia transportation, for example, the power issue becomes more important. As with the power limitations, network throughput also becomes an issue if more is demanded of the sensor network.

As the technology matures, it can be expected that these devices will get smaller and smaller, and become more capable at the same time. Many applications for these devices have yet to be discovered.

A. THESIS MOTIVATION

Military operational requirements drive the development of new technologies and modification of current technologies to meet applicable mission objectives. The latter is the more desirable of the two as it usually requires lower developmental costs to come up with a working solution. By looking at sensor networks in this light, it is possible they may provide capabilities outside of what they were originally designed for.

Sensor networks were designed to incorporate small, lost cost, portable devices that could collect and report on physical or environmental conditions. This type of information reporting suits these devices well as it requires less power and lower bit rates. It is outside this base area of operations that the aim of this thesis is directed.

The warfighter of today is more reliant on communications than ever before. Command and Control of military forces relies daily on effective communication from the command center level down to the lowest echelon soldier in the field. While sensor networks may initially play a small role on the battlefield, increased hardware capability may open up new applications for these networks.

By configuring the sensor nodes as repeaters, it may be possible to use the devices, on a limited basis, as a means to extend network communications into regions normally difficult to access. Their small profile and portability would allow them to be nearly invisible to the enemy, meanwhile serving as a pipe for vital communications to forces in theater. While limited in power and throughput, they may provide a temporary solution where no others exist.

B. THESIS OBJECTIVE

In order to utilize sensor networks as a bridge between TCP/IP networks, it is necessary to look more closely at how sensor networks work and decide what measures must be taken to facilitate this implementation. TCP/IP traffic is based upon the principle of assuring that the transmitted packets will be received at the desired location. Sensor networks are normally passing information considered non-vital and thus handle the data accordingly. On top of that, when working with a wireless medium, a large amount of loss can be expected. In order to consider using a sensor network to pass TCP/IP traffic, particular emphasis must be placed on data reliability across the network.

Two means of providing better reliability were considered. First, using Automated Repeat Request, or ARQ, was analyzed. Considerable research has already been done in this area. While it would be an effective means of verifying whether or not packets were received properly, the difficulties of employing this method with multiple sensor nodes were daunting. The work that this thesis was based on used a simple broadcast of packets that each node repeated until received by the destination. Changing this scheme would present two major problems [1]. By adding a feedback channel, the throughput would be significantly decreased. If the end goal was to be, for instance, passing Voice over IP (VOIP) traffic over the network, throughput would be a major concern. Something in the order of 90 kbps would be required for a reliable channel depending on the codec chosen [2]. Second, the use of an ARQ response would be fairly straightforward when using a simple, one-hop network, but would increase exponentially in difficulty as more nodes were added to the topology.

Because of the difficulties of using acknowledgements, Forward Error Correction (FEC) was chosen to add reliability to the communication path. Throughput was a concern but the implementation was considerably easier. Based on the FEC schemes available, a form of block coding was chosen for the experiment. The implementation of the block coding scheme and the results of its implementation are discussed later.

The overall objective of this thesis was to achieve the following:

1. File Transfer Capability

The preceding work on this project successfully set up a Java based implementation of a text messaging service across a sensor network. Messages that were typed in to the sending terminal were packaged into packets and sent over the network to the destination terminal which assembled the packets and displayed the message. To improve the capabilities of the channel, the next step was to add the ability for transporting files across the network. Specifically, the ability to send Joint Photographic Expert Group (JPEG) encoded images was desired. Adding this capability would improve the usefulness of the channel and make it possible to evaluate the effectiveness of an FEC algorithm. The smaller text based messages would not provide enough packets to make a proper evaluation.

2. Implement Forward Error Correction to Improve Reliability

Adding FEC to the channel would improve the reliability but also decrease the throughput. It was necessary to look at the effectiveness of the chosen algorithm and the throughput that resulted from its implementation. During the experiment, the FEC implementation would be compared against two different transmission schemes. One would be transmitting the data from a representative JPEG image without any redundancy. The second would be transmitting a copy of the packets along with the original packets. This second scheme would send roughly the same number of packets as the FEC scheme and provide a better basis for comparison.

3. Provide Recommendations for Follow-on Work

This thesis is only a small step towards the goal of being able to send TCP/IP traffic across a sensor network. Once the analysis of the preceding steps is complete, recommendations for possible follow-on work will be provided. Lessons learned from the experiment will be used to form the recommendations. Related work in sensor networks will also be considered while making the recommendations.

C. RELATED WORK

Sensor networks, as do traditional wireless networks, experience considerable challenges when it comes to providing reliable network communications. Hardware limitations, network configurations, and the environment all play a role in increasing the difficulty for networks to pass information efficiently. There have been a number of studies in the area of improving sensor network reliability. This thesis pulled ideas from many of these studies to either aid in the experimental phase or to gather information in order to form a proposal for future recommendations. Research in error correction and Medium Access Control (MAC) were particularly useful. Also this thesis drew heavily from previous thesis topics that covered the areas of erasure coding and tunneled networks.

1. Forward Error Correction in Wireless Sensor Networks

Several useful studies of error correction in sensor networks were found while doing research for this thesis. Students at the University of California Berkeley conducted tests of single and double error correcting codes in outdoor and indoor tests [3]. Another interesting research topic involved using an adaptive FEC code control algorithm for sensor networks [4]. In the study, they identified the need for something other than fixed correction codes for channels with constantly varying bit error rates. Finally, Terry Norbraten’s work with erasure codes and detailed explanation of the Java FEC Library from Onion Networks were extremely helpful during this research [5].

2. Medium Access Control for Wireless Sensor Networks

One of the realizations after examining the results from this thesis was that simply using error correction alone to improve the reliability of a channel is not sufficient. With increasing network topology complexity, additional measures should be considered. One such measure deals with modifying medium access control (MAC). A few different protocols were examined. Z-MAC is an exciting MAC protocol that achieves high efficiency by acting as hybrid between TDMA and CSMA [6]. Z-MAC behaves like CSMA during periods of low contention and like TDMA during periods of high contention. By using this approach, it aims to maximize efficiency during all phases of network activity. Two other protocols, T-MAC [7] and S-MAC [8], represent hybrids between TDMA and CSMA, although these protocols put more emphasis on energy efficiency while Z-MAC aims to maximize network throughput.

3. Tunneled Data Transmission Across Sensor Networks

Last but not least, the thesis work of Yow Thiam Poh on the topic of tunneled data transmission over wireless sensor networks was the foundation that this thesis was built upon. Yow created a text messaging channel across a sensor network and tested the channel’s efficiency with varied configurations and varying transmission parameters [9]. The goal of this thesis was to make the channel, created by Yow, more capable by adding file transfer and error correction.

D. THESIS ORGANIZATION

Chapter I presents the motivation and objective of this thesis. It also aims to provide the reader with a general idea of operational applications, outside of environmental sensor reporting, that are possible with wireless sensor networks.

Chapter II takes a look at wireless sensor networks and how they might be used as a bridge for TCP/IP networks. Covered in the chapter are the capabilities and limitations of the sensor networks and how the limitations might be mitigated. Also, detailed descriptions of the modifications that were made to Yow's text messaging channel in order to add file transfer capability are provided in this chapter.

Chapter III covers the use of FEC to enhance the performance of sensor network transmissions. A brief overview of the different FEC methods available is given. The proposed method of FEC to be used in the experiment, erasure coding, is covered more thoroughly. Also, how the FEC code is to be implemented is described in detail.

Chapter IV begins by presenting the setup of the experiment. The hardware and software used are described in detail. Following the setup discussion, the experimental procedures and results are presented.   The chapter concludes with an analysis of the results.

Chapter V provides the conclusions resulting from the experiment. In this chapter, future work and recommendations will also be proposed.

THIS PAGE INTENTIONALLY LEFT BLANK

II. WIRELESS SENSOR NETWORKS

A. OVERVIEW OF WIRELESS SENSOR NETWORKS

Originally motivated by military applications like battlefield surveillance, wireless sensor networks have rapidly become a growing industry with applications for both civilian and military interests. These networks are made up of spatially distributed autonomous devices that incorporate sensors to monitor physical and environmental conditions. The individual devices, or motes, are made up of a radio transceiver, a microcontroller, and batteries for a power source.

Sensor networks form ad-hoc networks upon deployment, allowing the transfer of sensor data to a parent node where the data is collected. A few examples of possible applications are battlefield condition reporting, home automation, and traffic control. Eventually, this new technology may affect all aspects of typical daily lives.

1. Devices

The functions that are typical of wireless sensor devices are communication, computation, and sensing. In order to perform these tasks, these devices incorporate a microcontroller for computation, a small amount of RAM for dynamic data, one or more flash memories that store the program code and long-lived data, a wireless transceiver, an antenna, an analog-to-digital converter (ADC), single or multiple sensors, and a power source [10]. Many variations of these devices are available. For example, a variety of power sources is possible. Batteries, solar power, and external power are all being utilized depending on the type of deployment. Also, some devices feature multiple processors in order to incorporate a digital signal processor (DSP), although these chips tend to consume greater amounts of energy. As the hardware capability improves and additional applications are targeted, the number of variations can also be expected to increase.

2. Operating System

The operating system for wireless sensor network devices tends to be a very simple design that handles interrupts and performs simple scheduling of tasks. While research has introduced a number of operating systems for the emerging technology, TinyOS continues to be the most popular. TinyOS is an open source operating system that started as a collaboration between the University of California, Berkeley and Intel Research. Since that time, it has grown into an international consortium called the TinyOS alliance.

TinyOS is a component based operating system written in nesC, a C based programming language. Components for certain tasks, such as packet communication, routing, sensing, actuation, and storage, are connected together using interfaces. Since development of the first TinyOS platform in 1999, many releases have been developed up to the current release of TinyOS 2.1.0 in August of 2008.

3. Communications Stack

Wireless Sensor Networks are typically multihop networks and rely on a communications stack that includes, medium access control (MAC), routing, and transport layers. Many available protocols exist for these layers and they differ from those used in wired networks and Wi-Fi networks. This is due to the different environments where these devices are located and the additional constraints that they must adhere to. Many limiting factors drive the need for specialized MAC, routing, and transport layers. Among them, small amounts of memory make large routing tables impossible and energy limitations limit communication ranges. Figure 1 depicts a typical WSN stack.

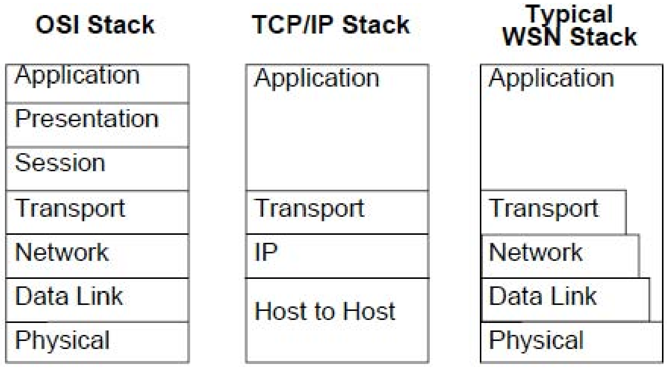


Figure 1. Comparison of the OSI, TC/IP, and Typical WSN Stack

An interesting feature of the WSN communication stack is self-organization. A few different methods of self discovery are utilized. In one example, devices discover their neighbors and append this information to a neighbor table. These tables include such information as node identification and geographic location. Using this location information, devices can then perform routing. Another example has the base station learning the entire network topology, using it to create a spanning tree routing structure [10].

B. WIRELESS SENSOR NETWORKS AS A TCP/IP NETWORK BRIDGE

An exciting application worth considering is the temporary use of sensor networks as a bridge for TCP/IP networks. Although throughput and power limitations prevent these devices from performing more intensive data transfers, temporary use to aid the warfighter in difficult environments may be worthwhile.

1. Benefits of Implementation

Wireless network communications are becoming more prevalent for military operations. Intelligence reports, imagery, and general communication are reaching further into the battlefield than ever before. Typically, the infrastructure found in these environments is limited or non-existent.

Wireless sensor networks were originally designed for the purpose of reporting on the environmental and physical conditions of the battlefield. Using these networks to temporarily extend vital TCP/IP network communications might be possible despite the limitations of the sensor nodes. A few specific applications of interest are Voice-Over-IP (VOIP) and the transmission of time sensitive data and imagery. In order to consider this as a possibility, the difficulties involved must first be considered

2. Barriers for Implementation

Sensor networks were designed to transmit small amounts of data with limited frequency. As a result, the sensor nodes typically have limited onboard memory. The MICAz motes used in this research have 128 kbytes of flash program memory, 512 kbytes of flash log memory, and only 4 kbytes of RAM. Since the data they transmit is considered non time-sensitive and non vital, the networks do not have to incorporate many of the assurances necessary in TCP/IP networks. Tunneling over the sensor network may involve increasing the onboard memory to support an enhanced network stack or finding ways to more efficiently use the smaller amount of memory.

Power is another limitation that must be addressed. As sensor motes were designed for low power, autonomous operation, batteries or solar power are typically employed. Increasing the data amounts and rates would increase the power demand and threaten the longevity of the device.

In order for this application of sensor networks to be successful, the throughput of the network would have to provide a certain level of performance to meet the needs of the user.   To use the network to pass VOIP traffic, for example, a minimum data rate of about 90 kbps would have to be supported to provide adequate communications. For transmission of time sensitive data or imagery, throughput would have to meet specific mission requirements.

By testing the transmission of image files across a sensor network during the course of this research, it was desired that a general idea of the throughput capabilities would be found. Although the forward error correction implemented would reduce the channel's throughput, it was considered necessary for the channel's reliability.

C. CONTINUING DEVELOPMENT OF WIRELESS SENSOR NETWORK ADAPTATION INTERFACE LAYER (SNAIL)

As mentioned before, the experiment portion of this thesis was based on the work of Yow Thiam Poh, a former student at the Naval Postgraduate School. His work involved the development of a sensor network channel that allowed a form of text messaging. He developed three Java applications that would work in coordination with the sensor mote hardware from Crossbow to achieve this task. The suite of applications was referred to as the Sensor Network Adaptation Interface Layer (SNAIL). SNAIL consisted of separate Client and Server modules that were used on a transmitting laptop, and a Listen module that was used on the receiving laptop. These SNAIL modules were modified extensively to achieve the goals of this thesis.

In order to incorporate a more robust channel by adding error correction, the first step was in modifying the SNAIL software to incorporate file transfer. The larger amounts of data associated with file transfer would allow the improvements provided by error correction to be observed. The file types that were chosen for implementation and testing were JPEG images, test files, and MS Office documents. These were chosen due to their popularity and everyday use. How each of the SNAIL modules was modified is explained below.

1. SNAIL Client Module

The SNAIL Client module now presents the user with three options upon running the application. Choices are now for text message transfer, standard file transfer, or JPEG image transfer. Before passing off the data to the SNAIL server module, a few modifications were necessary.

Previously, the Client module allowed the user to input a text message from the terminal to be transferred over the sensor network. Using a blocking reader module, upon receiving a message from the user, the message was read in as a string and then converted to a character array. The use of a character array was chosen to allow some flexibility with the data stream. Essentially, this allowed adding the user’s selection to the data stream before passing it on to the Server module. Prior to the IO operation, the character array had been converted back to a string.

To accommodate the transfer of files, the first modification was changing the module to work with byte arrays. Should the user choose to send a text message, the message is still read in as a string, but is converted to a byte array. The option selected is added to the byte array by using concatenation of arrays. By using byte arrays, the data is now compatible with the file transfer options. Strings could have been used for both, but ultimately that would have limited the file length to roughly 64 kbytes.

Upon the choice of either standard file transfer or JPEG image transfer, the user is presented with a file selection box. Upon selecting the desired directory and file, the file data is read in to a byte array. The two file transfer options are executed differently. For standard file transfer, a fileinputstream is opened to pull in the data to a byte array. To pull in a JPEG image correctly, the Java ImageIO tools were used to read in the file as a bufferedimage. After the file is read in, it is converted to a byte array.

By utilizing byte arrays for each of the three options, all three are assured to be compatible and the same flexibility to modify the data stream that character arrays afforded is maintained. Finally, the data residing in the byte array is transmitted to the SNAIL Server module via an ObjectOutputStream. Figure 2 presents the process decisions of the SNAIL Client module as a flow diagram.

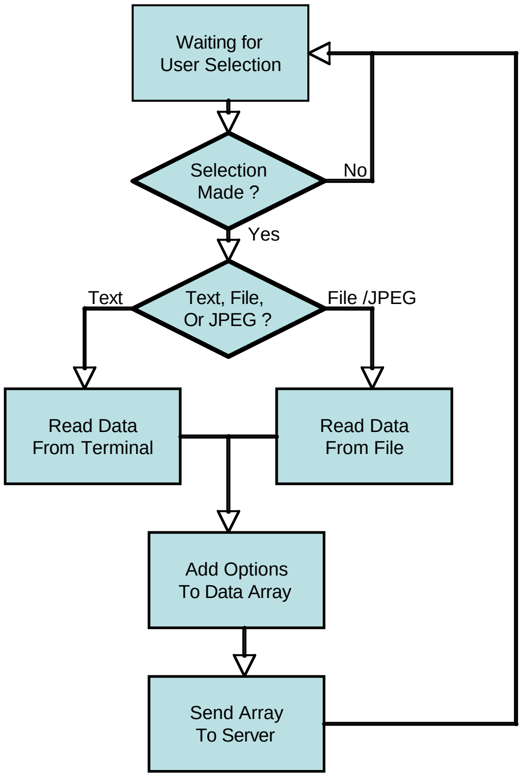


Figure 2. SNAIL Client Flow Diagram

2. SNAIL Server Module

The SNAIL Server module required more extensive modifications to allow file transfer and error correction. Error correction will be covered later. For now, data handling and packetizing for sensor network transport will be covered.

After reading the data sent from the SNAIL Client module, the data is stored in a byte array. One of the main changes is the handling of all data as byte arrays from start to finish. From the data array, the option that was selected is obtained. The option will ultimately be removed from the data and transmitted as part of a handshake packet. The handshake packet is the first packet that will be transmitted by the server and contains much of the amplifying information needed by the receiver.

Header(5)	Payload(29)	CRC(2)

Figure 3. Packet structure of TinyOS (from [8])

Should encoding not be desired by the user, the number of packets to be sent is calculated from the array size. The array size is a key piece of information needed by the receiving terminal and is included in the handshake packet. Of the 29 bytes of data available in the Active Messaging (AM) packet, two of the bytes will be used for Terminal ID and packet number for this non-encoding case. Figure 3 shows the standard AM packet structure used by TinyOS. The packet number will help the SNAIL Listen module keep track of which packets have been received and facilitate dropping redundant packets.

The use of the header packet was changed only with regards to content. Because of the new error correction option, additional information was required at the receiver. One new addition to the transmission process was the use of a terminating packet. After all of the data packets are sent out, the terminating packet is sent out which contains an identifying byte sequence. This packet was added to correct the condition where packets are dropped and the receiving end is stuck in a loop waiting for packets. Once the terminating packet is read in, the Listen module is free to move on to analyzing the data.

Figure 4 presents the flow diagram for the SNAIL Server module.

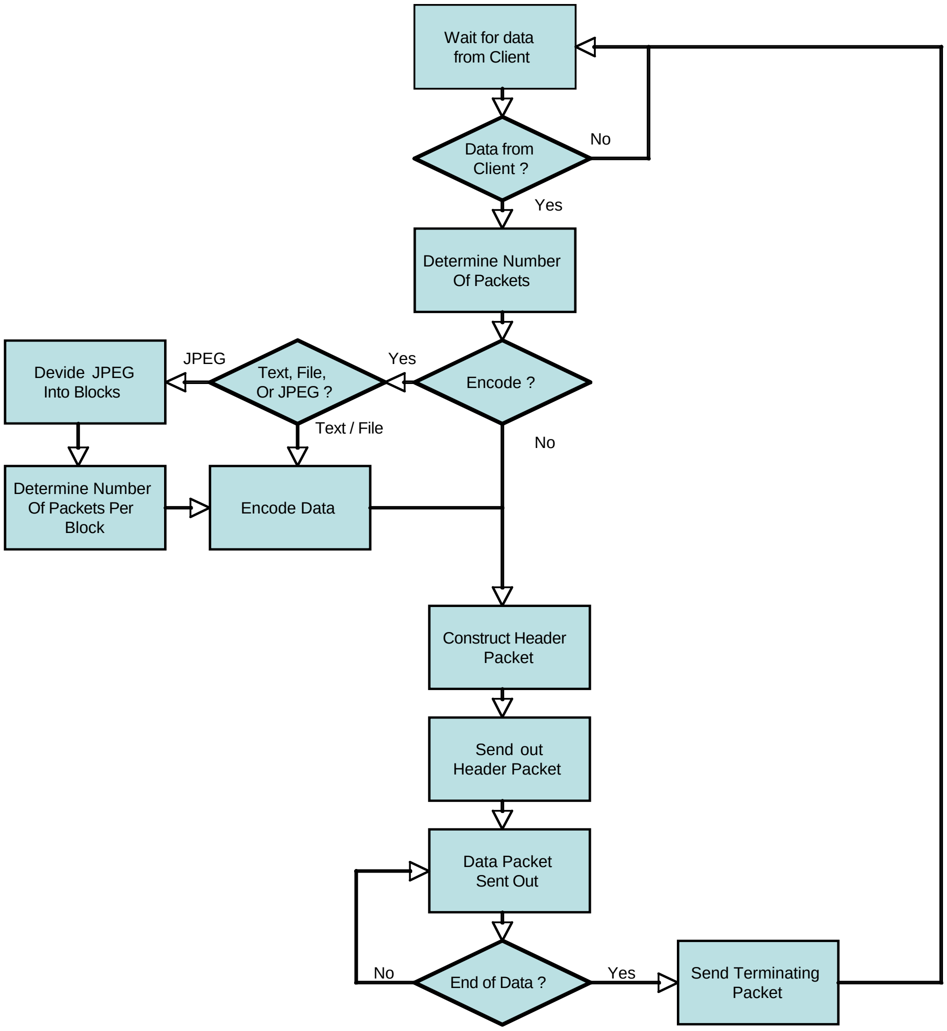


Figure 4. SNAIL Server Flow Diagram

3. SNAIL Listen Module

Upon receiving the handshake packet, the SNAIL Listen module knows whether or not the data will be encoded, what option was selected, and the length of the data array involved. Using the array length, the number of packets to be expected is calculated.

A loop is entered in which each of the packets is read in. Both the handshake and terminating packets contain flags to help identify them from normal packets. Also, as the packets are read in, the packet number is obtained which is used to identify the packet. If the packet is redundant, it is dropped. A change was made to help keep track of which packets are read in. Previously a packet counter expired when the expected number of packets was reached. This was changed due to the encoding option. This will be covered further in the next chapter.

Once all of the data has been read in and decoding has been completed, if necessary, the option selected determines how the data is handled. If a text message was selected, the message is displayed on the terminal. If a standard file was transferred, the file is saved to the location chosen by the user using a fileoutputstream. For a JPEG image, the byte array is converted back into a bufferedimage and then the image is stored using the ImageIO utilities.

Figure 5 presents the flow diagram for the SNAIL Listen module.

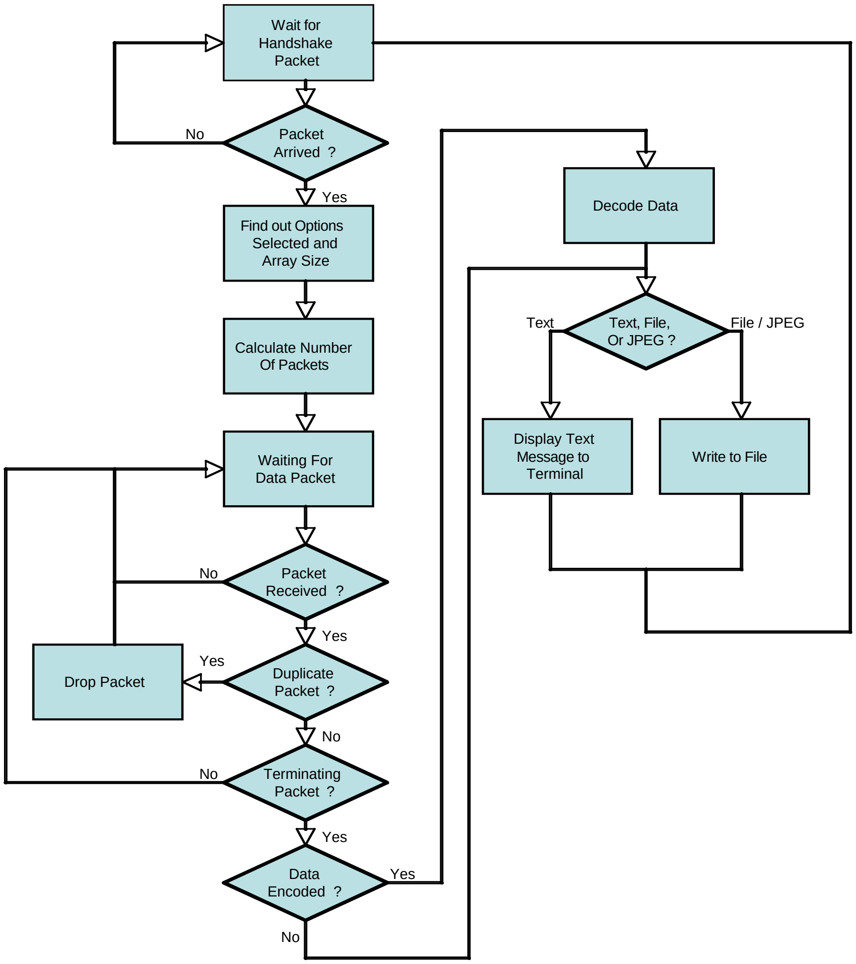


Figure 5. Flow Diagram for SNAIL Listen Module

D. SUMMARY

Wireless Sensor Network design involves even greater complexity than traditional wireless networks. Due to the limitations of the devices, greater emphasis on MAC, routing, and transport layer design is required to combat the typical congestion and losses associated with data transfer in wireless networks. Despite the challenges, many new and exciting applications are being created to take advantage of the size and portability of the devices. Possible applications like that of TCP/IP data transfer across sensor networks may soon become a reality. This chapter concluded with an update to previous thesis work in the area of transporting data messages across sensor networks. By adding file transfer capabilities, studies regarding forward error correction’s effects upon sensor network performance can be pursued.

THIS PAGE INTENTIONALLY LEFT BLANK

III. FORWARD ERROR CORRECTION IN WIRELESS SENSOR NETWORKS

In wireless networks, packet loss is inevitable. In order to combat this packet loss, either Forward Error Correction (FEC) or Automatic Repeat Request (ARQ) or a combination of the two techniques are used. ARQ is an attractive option since it is relatively inexpensive in that it requires no manipulation of the data being transferred. Under conditions of increasing losses, ARQ does suffer significant reductions in overall throughput. Tradeoffs exist between complexity of implementation and data throughput. ARQ also rapidly becomes more complicated as the number of clients grows.

FEC, on the other hand, detects and corrects losses incurred by a noisy channel by including redundant information with the data it passes. This has the advantages of allowing the correction of errors more quickly than with ARQ and by simplifying the network traffic scheme. Some kind of feedback channel could be included but may not be necessary.

A. GENERAL OVERVIEW OF AVAILABLE FEC CORRECTION METHODS

The available types of FEC are broken down into two categories. These are block codes and convolutional codes. Block codes work on fixed-size blocks of bits or symbols of a fixed size. Convolutional codes work on bit or symbol streams of various sizes.

1. Block Coding

Many different forms of blocks codes exist. A few examples are the Hamming code, BCH code, and Reed Solomon code. The latter is the most widely used due to its near optimal coding qualities.

A Reed Solomon code encodes a data message block as points in a polynomial function plotted over a finite field [11]. The polynomial coefficients are the data symbols of the block. These codes can work to correct errors at either the bit-level or the packet level. Lost data packets are corrected from encoded packets, otherwise known as repair packets. These repair packets represent a set of linearly independent equations. By solving this set of equations, the lost packets are recovered. One drawback to the Reed Solomon coding scheme is encoding and decoding time which is $O(n^2)$ and $O(n^3)$ respectively. Another is the memory requirement resulting from the polynomial operations.

2. Convolutional Coding

Convolutional coding involves taking an *m*-bit message that will be encoded and converting it to a *n*-bit symbol. The code rate for the encoding process is m / n where $n \geq m$. The constraint length of the code, *k*, determines the error correction capability and the complexity. As *k* increases, the correction capability increases but the complexity also increases exponentially. For decoding convolutional codes, the Viterbi algorithm is commonly used. The Viterbi algorithm uses maximum likelihood estimation to make decisions regarding the underlying probability distribution of the bits received [12]. For effective correction, a constraint length of at least 7 and typically below 9 is used while a code rate m / n of at least 1 / 2 is required. A convolution code has reduced complexity over a Reed Solomon code but suffers higher coding redundancies. For this reason, convolutional codes are more ideal for communication channels with a lower signal-to-noise rate (SNR). A convolution code is also not typically used for the recovery of lost packets as is Reed Solomon.

B. ERASURE CODING

For this experiment, an implementation of erasure coding was chosen. Erasure coding is basically a form of block coding that takes a number of data packets, or blocks, and encodes them into a larger number of encoded data packets. The larger the number of encoded data packets, the more redundancy allowed. As long as a minimum number of the transmitted packets reach the destination, the source data can be reconstructed. The key to erasure coding is that the destination knows exactly which packets have been dropped. Without that knowledge, this coding scheme would not work.

Erasure coding was chosen due to the inclusion of CRC in the wireless sensor network packets. Since the lower layers of the protocol stack would check arriving packets for errors, only allowing error free packets to reach the application layer, this method of FEC coding seemed appropriate. Packets lost in transmission or dropped due to errors would not prevent the successful transmission of the source data. An open source JAVA implementation of erasure coding created by Onion Networks was implemented during the experiment. Based on the work of Rizzo [13], this Java based software library will be discussed further in the next section.

1. Erasure Coding Fundamentals

The basis behind erasure coding is that *k* blocks of source data are encoded producing *n* blocks of encoded data [13]. If any subset of the *n* encoded blocks is received at the destination, the receiver is able to reconstruct the source data. This code is referred to as an (*n*, *k*) code. In this scheme, up to *n* – *k* losses are acceptable. Figure 6 gives a graphical representation of the source data encoding and reconstruction.

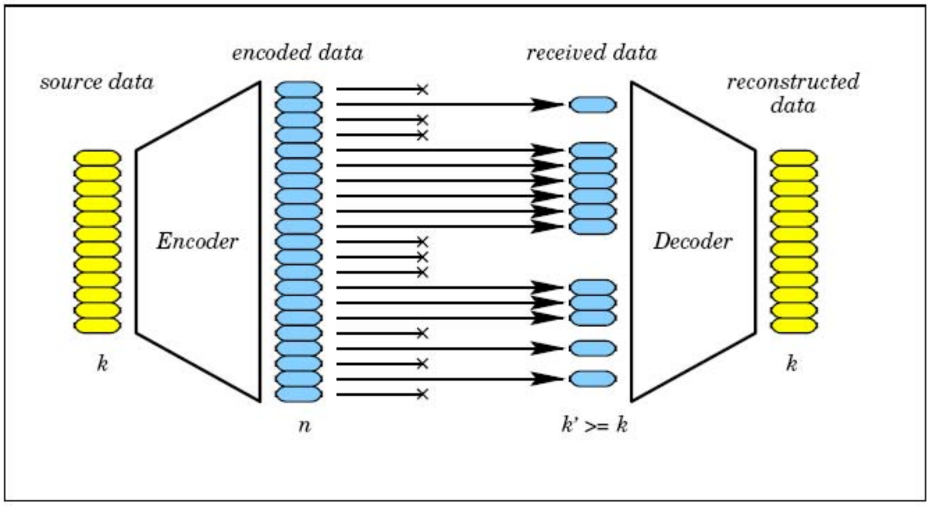


Figure 6. Data Reconstruction in Erasure Coding (from [13])

A subset of the erasure codes, called linear codes, can be analyzed using the properties of linear algebra. If $\underline{x} = x_0 \ldots x_{k-1}$ represents the source data and G is an $n \times k$ generator matrix, then $\underline{y} = G\underline{x}$ is the (n, k) linear code resulting from the matrix multiplication. As long as k components of $\underline{y}$ are received, $\underline{x}$ can be recovered.

If the encoded data contains an exact copy of the source data, this is referred to as a systematic code. With a systematic code, a portion of the generator matrix, G, will contain the identity matrix. Systematic codes can be very advantageous if very few losses are expected in the link. Reconstruction of the source code would be greatly simplified. Figure 7 once again shows the encoding and decoding process but this time in matrix form for a systematic code. Note the identity matrix in the upper portion of the generator matrix.

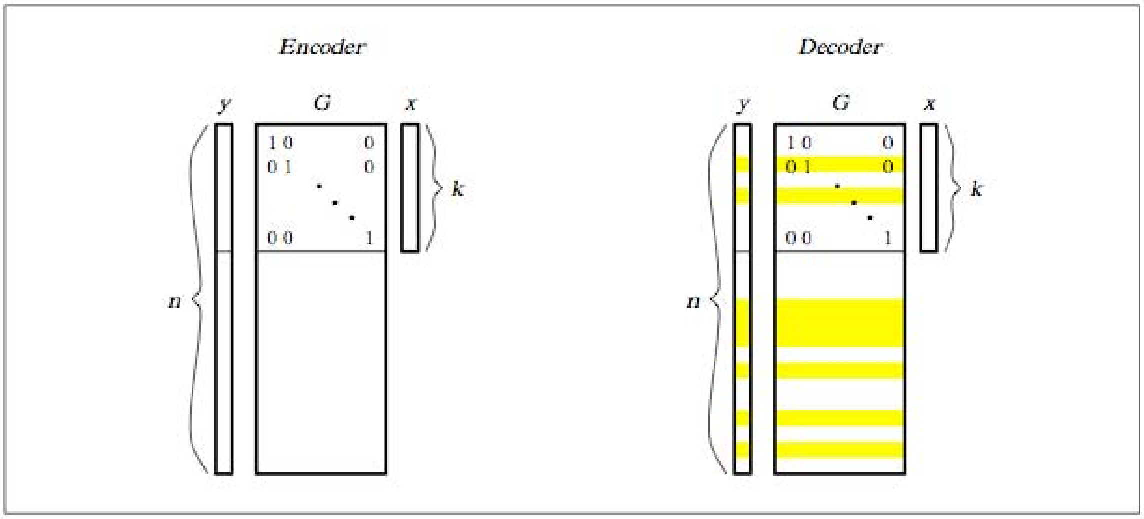


Figure 7. Encoding / Decoding Systematic Code (from [13])

The generator matrix G is a $n \times k$ matrix of rank k. Because of this, only k of the n encoded packets are necessary. Each column of G can be composed of a maximum of $(k - 1)$ nonzero elements. For the systematic code example, since the columns already have $(k - 1)$ zero elements due to the identity matrix, all of the remaining elements are required to be nonzero.

For the reconstruction process, along with the encoded packets of data, the identification of those packets must also be known. This will add overhead to the process as the transmitting end will have to include this information with the transmission. This is a negligible amount of overhead though and the packet identification will also aid in with identifying redundant packets so they may be ignored. The recovery is performed by solving the linear system:

$$\underline{y}' = G'\underline{x} \rightarrow \underline{x} = \left(G'\right)^{-1} \underline{y}' \qquad (1)$$

For the above equation, $\underline{x}$ represents the original source data and $\underline{y}'$ is a subset of k encoded packets. *G'* is the corresponding subset of columns from the generator matrix. To reconstruct the original data, the inverse of G' is taken and then multiplied by the subset of encoded packets, $\underline{y}'$. The cost of inversion is somewhere in $O(kl^2)$, where $l \leq \min(k, n-k)$. The value of l represents the minimum number of packets that must be received.

2. Erasure Code Based on Vandermonde Matrices

An example process for the creation of a generator matrix can be shown with the use of a Vandermonde matrix. This matrix has coefficients of the form

$$g_{i,j} = x_i^{\,j-1} \qquad (2)$$

where the x_i ’s are elements of extension fields, or $GF(p')$.

Extension fields are a subset of finite fields that allow basic arithmetic to be performed on data much like it is done with integers. They help resolve problems associated with handling the number of bits needed to represent the result of computations. Mapping data elements into field elements prior to arithmetic operations and then applying the reverse mapping to get the desired results avoids this trouble. If finite fields are not used and the results of the coding arithmetic operations are rounded prior to transmission, exact reproduction of the data would not be possible.

Seen in matrix form, the $n \times k$ Vandermonde matrix is

$$G = \begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{k-1} \\ 1 & x_3 & x_3^2 & \cdots & x_3^{k-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{k-1} \end{pmatrix}. \tag{3}$$

The determinant of a square Vandermonde matrix is defined as

$$\prod_{i,j=1\ldots k, i<j} (x_j - x_i). \tag{4}$$

The matrix will have a non-null determinant and thus be invertible if all of the x_i 's are different. As long as *q* > *k* and all x_i 's are not equal to zero, *q – 1* rows at a maximum can be created, where *q* is the number of finite field elements. If the identity matrix is added, a suitable generator can be created for a systematic code.

Considering a few special cases for the code, a (*n*, *1*) code would simply create copies of the single packet. This is essentially the same thing as making multiple copies of the same packet to be sent out. The work that this thesis built upon utilized this simple method of improving the link performance by sending multiple packet copies. Unfortunately, this type of code is inefficient compared to codes with higher values of *k*. A (*k+1*, *k*) code is another simple case. This would include the *k* packets plus one packet that would represent the sum of the others. Once again, this case is not very useful except for channels with small amounts of loss.

3. Erasure Codes for Wireless Sensor Networks

Once again, erasure codes were chosen to combat the relatively high amount of packet loss that can be expected with wireless sensor networks. While the software implementation of erasure codes is somewhat computationally expensive, low to medium speed applications, up to the 100 KB/s range, could be supported with fairly low amounts of overhead. Given the limitations of power and throughput with these networks, erasure codes may be a very useful tool. Combining this technique with a simple form of ARQ might be the best course of action.

C. PROPOSED SOLUTION

1. Onion Networks JAVA FEC Library

While investigating Rizzo’s C implementation of the Vandermonde based erasure codes, a Java based version of the same was discovered. Onion Networks, Inc. developed this open source Java based implementation they refer to as the FEC 1.0.3 library. Considering the SNAIL applications were developed in Java, this seemed a logical fit and worth exploring how they could be used together.

The FEC library incorporates a number of classes and tools that can be used to encode and decode data given the specifications of *k* and *n*, the number of packets to encode and the number of encoded packets to create. The library supports a pure Java implementation of the code as well as a C based implementation that can achieve higher speeds. For this experiment, the Java based implementation was chosen for compatibility reasons.

2. SNAIL Server Modifications

The Java code below shows the actual encoding process of the data.

```
//  FEC Setup Procedure  //

byte[][] repair = new byte[numblocks][n*packetsize];
//this is our encoded data

int[][] repairIndex = new int[numblocks][n];

//These buffers allow us to put our data in them they
//reference a packet length of the file (or at least will once
//we fill them)

//create our fec code
if (encoded == 0)      // I only want to do this section if we
are going to encode
{
  FECCode fec = FECCodeFactory.getDefault().createFECCode(k, n);
  // creating code

  for (int w=0; w < numblocks; w++)
  {
     Buffer[] sourceBuffer = new Buffer[k];
     Buffer[] repairBuffer = new Buffer[n];

     for (int i = 0; i < sourceBuffer.length; i++)
```

```
                sourceBuffer[i] = new Buffer(source2[w], i*packetsize,
…               packetsize);

             for (int i = 0; i < repairBuffer.length; i++)
                repairBuffer[i] = new Buffer(repair[w], i*packetsize, …
                packetsize);

             //When sending the data you must identify what it's index
             was.

             for (int i = 0; i < repairIndex[w].length; i++)
                repairIndex[w][i] = i;


             //encode the data
             fec.encode(sourceBuffer, repairBuffer, repairIndex[w]);
           }
         }
         //  End FEC Encoding //
```

Prior to encoding, the values of *k* and *n* must be chosen. For the experiment, 64 and 128 were chosen respectively. From Rizzo's analysis of erasure coding, it was explained that increasing values of *k* improved efficiency. From early experiment tests, this improvement was seen in increased values of *k* up to a certain point. Also, as *k* is increased, the number of blocks the image is broken up into is decreased. Increasing or decreasing the number of blocks has a negligible effect on total encoding time.

Looking at the code, the two byte arrays, source and repair, hold the source data and encoded data respectively. The sourceBuffer and repairBuffer buffers are used to temporarily hold the data in packet form needed for the encoding process. The repairIndex array holds the numbering of each of the packets to be encoded. The encoded packet number must be transmitted with each packet in order to reconstruct the data. The last line of code calls the encode method which completes the encoding process, storing the now encoded data into the repair array.

The SNAIL Server module now transmits the data as it would with non-encoded data. The only differences being the addition of the block number and roughly double the number of packets for the chosen *k* and *n*.

3. SNAIL Listen Modifications

Modifications were necessary for the SNAIL Listen module to receive encoded packets. As mentioned before, the block number and packet number are sent in each packet. Also the handshake packet contains the encoding parameters *k* and *n* that were chosen. By not hard coding this information, experiments with differing values of each are possible.

Prior to the modifications to the SNAIL Listen module, packets were read in until the number of expected packets had been reached. If the number of packets expected was not reached, it was considered a failure to transfer the text message. With error correction, this is not the case. In order to keep track of what packets had been received, an integer matrix was constantly updated as packets were read in. The rows of the matrix represented the different blocks being read in and the columns represented the packets that each block contained. Initially the matrix was filled was zeros. As each packet is read in the respective matrix position was updated with a value of one. Once the terminating packet is received, if a particular block does not have at least *k* values read in, the data cannot be reconstructed.

```
// Decoding Process
System.out.println("Message Received: Decoding...");
byte[] received2 = new byte[arraysize];    // All data minus
                                           // padding
Buffer[][] receiverBuffer = new Buffer[numblocks][n];
// k subset of packets

for (int z=0; z < numblocks; z++)
{
   System.out.println("Decoding Block: " +(z+1)+" of " …
     +numblocks);
   //create our Buffers for the encoded data
   for (int i = 0; i < n; i++)
    {
       receiverBuffer[z][i] = new Buffer(received[z], …
           i*packetsize, packetsize);
    }

    //finally we can decode

    fec.decode(receiverBuffer[z], receiverIndex[z]);
  }
```

From the subset of code above, the decoding process is seen to be very similar to the encoding process. After all of the packets have been read in a check is performed to make sure enough packets are stored for each block. If this is the case, the decoding process commences one block at a time. Like during the encoding process, a buffer is used to hold the data for decoding and the result of the decode method is stored into a byte array.

D. SUMMARY

With any network, there are basically two methods for recovering erroneous packets. Either ARQ or FEC are used. Of the two, FEC was the preferred choice in this study due to power consumption limitations with wireless sensor networks and due to the complications of congestion when dealing with ARQ. A software implementation of FEC was preferred for this thesis given the Java based software platform that the original work was built upon. The FEC library by Onion Networks provided a very effective Java implementation of the erasure code written about by Rizzo. In the next chapter, the results of this implementation over a sensor network will be shown and compared to experiments without FEC.

IV. EXPERIMENT DETAILS

The performance of the link was the primary concern while conducting the experiments. Since a more robust link was desired, the performance of the sensor network link was tested with forward error correction implemented. In order to provide a comparison for the results, experiments without the FEC coding were also conducted. These results will hopefully provide a good foundation for what performance enhancements can be achieved using the FEC scheme selected.

This chapter aims to provide the results that were obtained from the laboratory experiment. Three different scenarios were chosen for conducting the test. These include a direct terminal to terminal test, a terminal to terminal test via one sensor mote hop, and a terminal to terminal test via two sensor mote hops. For each of these scenarios, three different data transmission schemes were tested. First, the erasure coding scheme was tested. The second scheme involved testing the link by sending only the image data across. No redundancy was used during this test. For the last scheme, redundancy was incorporated by sending a copy of each packet along with the original packets. This last scheme was chosen since it more closely approximates the total number of packets being sent out during the FEC test.

To test the three transmission schemes, a small 8 kbyte image was chosen to be transferred. The small file was chosen to allow for large number of tests to provide an adequate amount of results for a comparison. For a successful transmission, the entire image had to be transferred without error.

A. EXPERIMENT SETUP

The experiment was originally conducted in an academic building hallway to allow enough space to separate the base stations and motes in order to drive the link to the edge of its performance capabilities. By adjusting the power on the motes and investigating the resulting transmission range, the motes were placed to force a considerable amount of packet loss. This was necessary to prove that the FEC method would handle more challenging conditions better than the schemes lacking error correction.

After initial trials that were conducted, it was determined that multipath effects encountered in the hallway environment would negatively influence the results. Due to this realization, the experiment was relocated to an anechoic chamber which would help by eliminating a majority of the undesired signals. The chamber can be seen in Figure 8.

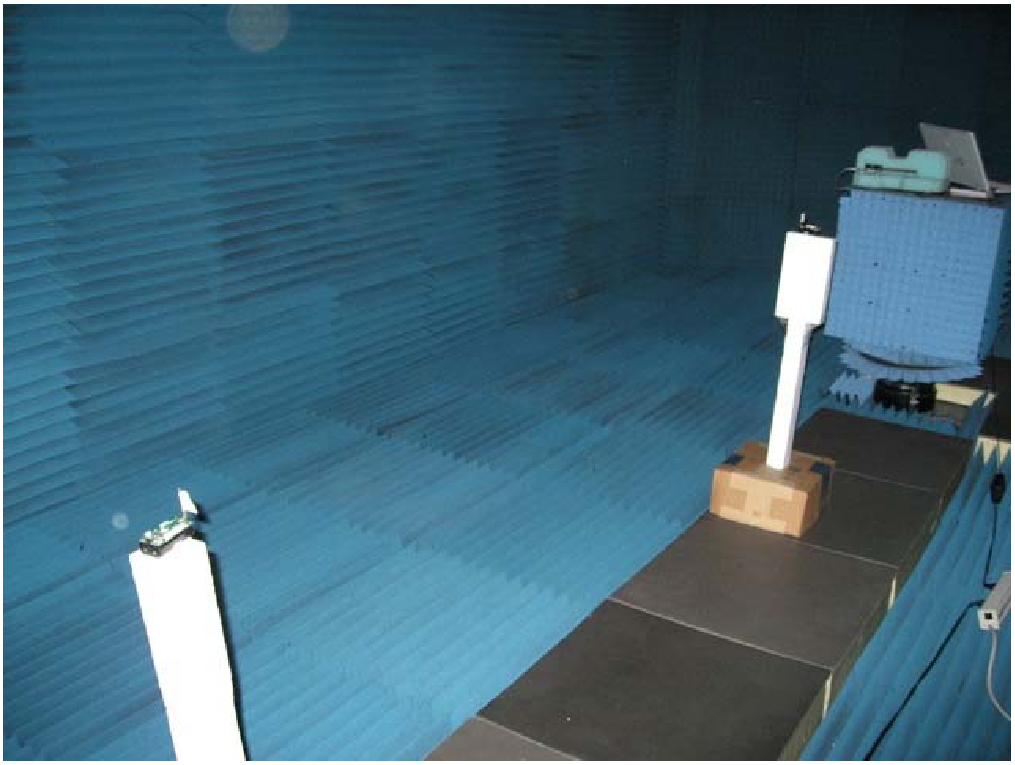


Figure 8. Anechoic Chamber used for the Experiment

1. Hardware

Each of the base stations incorporated a laptop and a Crossbow MICAz sensor mote connected through the USB port using a MIB520 sensor board. Individual MICAz motes were used for the repeater stations simulating the sensor network. Descriptions of each component used in this experiment are included below.

a. MICAz Sensor Motes

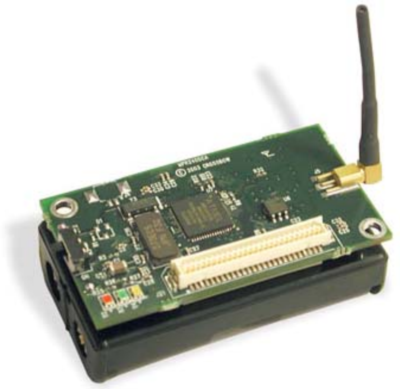
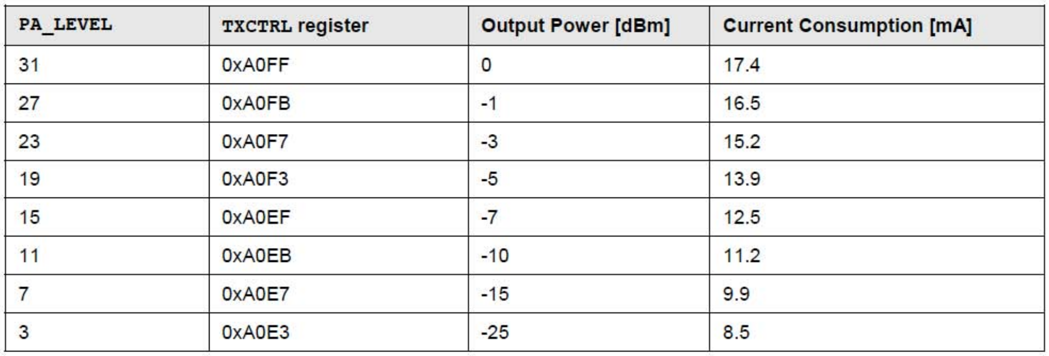


Figure 9. MICAz 2.4 GHz Mote

The MICAz mote, pictured in Figure 9, is a small wireless hardware device produced by Crossbow [14]. The mote has an Atmel Atmega microcontroller, 4 KB of RAM, 128 KB of program flash, 512 KB of flash log memory, a 2.4 GHz IEEE 802.15.4 transceiver that supports a maximum rate of 250 kbps, and a maximum range of 50 m. The mote also features a 10-bit ADC and runs on two AA batteries, drawing 8 mA in active mode.

The mote can be used with or without an optional sensor board providing capabilities as a wireless sensor platform or as a wireless node. A variety of sensor and data acquisition boards can be connected to the MICAz by means of a 51-pin expansion connector. The RF transmit power for the MICAz is user selectable from -24 dBm to 0 dBm. This feature was particularly helpful in allowing tests to be conducted in a small laboratory sized environment. The CC2240 Transmitter datasheet power levels are shown in Table 1. Experimental tests found that the power levels lower than those shown were also possible.

These MICAz motes implement a CSMA based protocol. This was determined by the CC2420 radio installed on the board and the version of TinyOS installed. A more advanced CSMA protocol, B-MAC, became available in the 1.1.3 version of TinyOS, but was not available for this experiment. Among the changes in this protocol was a variable noise floor over the fixed floor originally used. This noise floor is used in the determination process of when the mote can transmit. B-MAC's improvements on performance might be a valuable topic for further work which will be discussed further in Chapter V.



PA_LEVEL	TXCTRL register	Output Power [dBm]	Current Consumption [mA]
31	0xA0FF	0	17.4
27	0xA0FB	-1	16.5
23	0xA0F7	-3	15.2
19	0xA0F3	-5	13.9
15	0xA0EF	-7	12.5
11	0xA0EB	-10	11.2
7	0xA0E7	-15	9.9
3	0xA0E3	-25	8.5

Table 1. Available RF Power Levels for CC2240 Transmitter

b. MIB520 Parallel Programming Board

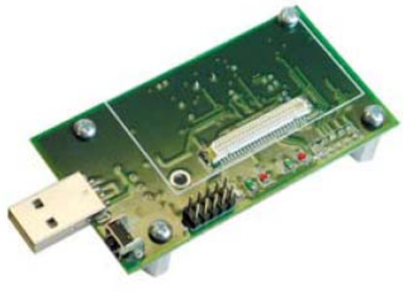


Figure 10. MIB520 Interface Board

The MIB520, pictured in Figure 10, provides USB connectivity to the MICA family of motes for in-system programming and general communication [14]. A MICAz node connected to the MIB520 can act as a base station. This configuration was used for the sending and receiving terminals configured for the experiment.

2. Software

Two software elements used for the experiment were Java and TinyOS. Java applications were written for the transmit side to read in, format, and encode the data for transmission. On the receive end, another Java application was written to read in the packets and process the information. TinyOS is one of the operating systems used by wireless sensor networks. For the experiment, TinyOS version 1.1.0 was installed on the laptops being utilized. TinyOS was written in nesC, a dialect of the C programming language. While most of the programming required for this thesis was in Java, some modification of nesC code was necessary for controlling certain parameters of the MICAz motes, such as the RF transmission power.

For the ease of programming, the Netbeans Integrated Development Environment (IDE) was used during the software development process. This software tool provided an eased integration of TinyOS and Java development.

B. TERMINAL TO TERMINAL EXPERIMENT

The first scenario for the experiment involved setting up a simple terminal to terminal link as pictured in Figure 11.

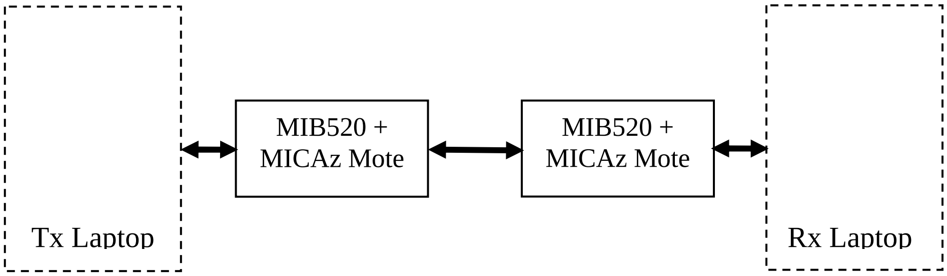


Figure 11. Scenario 1 – Terminal to Terminal Communication

This, the most simple of the arrangements to be tested, was used for the initial testing of the FEC code and would provide a good basis of comparison for the one hop and two hop tests to be conducted later. The Tx laptop shown in Figure 11 uses the SNAILServerTest_fec and SNAILClient_fec applications. Simultaneously running is the SerialForwarder application that is also used on the Rx laptop. In addition to SerialForwarder, the Rx laptop runs the SerialListenTest_fec application. Both MICAz motes were programmed with the TOSBase software. The TOSBase software was modified to set an RF power level of -24 dBm.

	Non-Encoded				Non-Encoded (Redundant)				FEC Encoded			
Run	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)
1	299	163	0	N	598	299	296	Y	640	623	0	Y
2	299	293	0	N	598	299	294	Y	640	637	0	Y
3	299	292	0	N	598	299	295	Y	640	616	0	Y
4	299	296	0	N	598	298	295	N	640	627	0	Y
5	299	286	0	N	598	299	291	Y	640	626	0	Y
6	299	296	0	N	598	299	283	Y	640	626	0	Y
7	299	294	0	N	598	299	299	Y	640	632	0	Y
8	299	286	0	N	598	298	276	N	640	635	0	Y
9	299	295	0	N	598	299	261	Y	640	620	0	Y
10	299	296	0	N	598	299	293	Y	640	633	0	Y
11	299	294	0	N	598	193	152	N	640	634	0	Y
12	299	292	0	N	598	299	286	Y	640	632	0	Y
13	299	246	0	N	598	237	100	N	640	636	0	Y
14	299	285	0	N	598	299	289	Y	640	630	0	Y
15	299	262	0	N	598	298	291	N	640	618	0	Y
16	299	272	0	N	598	299	297	Y	640	588	0	Y
17	299	287	0	N	598	299	297	Y	640	631	0	Y
18	299	287	0	N	598	209	67	N	640	614	0	Y
19	299	291	0	N	598	299	290	Y	640	493	0	N
20	299	289	0	N	598	298	249	N	640	466	0	Y
21	299	285	0	N	598	299	289	Y	640	593	0	Y
22	299	290	0	N	598	293	216	N	640	622	0	Y
23	299	291	0	N	598	295	249	N	640	389	0	N
24	299	293	0	N	598	299	279	Y	640	305	0	N
25	299	281	0	N	598	298	276	N	640	589	0	Y

Table 2. Terminal to Terminal JPEG Transmission Results

The transmitting and receiving motes were placed at a distance of 110 inches from each other. This distance was chosen to promote some loss of packets at the receiving end in order to test the effectiveness of the FEC algorithm. Looking at the results in Table 2, the FEC encoded transmissions were the best performing of the three tested methods. Of the twenty-five runs performed, the non-encoded scheme was unable to successfully transfer an image. Without any redundancy, losing a single packet constitutes a failure to transfer an image. Averaging 282 packets received out of the original 299 packets sent, this method of transfer did not provide for a robust link at the chosen distance.

The second transmission scheme sent a copy of each data packet or a total of 598 packets. This provides for the ability to lose random packets but consecutive lost packets could be a problem. This method proved successful for 60% of the transmissions, averaging 288 of the necessary 299 data packets. A counter was created to keep track of the extra or redundant packets that were received. An average of 260 redundant packets was received.

The performance enhancement of the FEC coding was obvious. On average 88% of the runs were successful; 589 of 640 sent packets were received on average.

Looking at run number twenty, the image was saved successfully even though only 466 of the 640 packets were received. The key of the erasure code algorithm is that for each block of image data being transferred, as long as *k* packets out of *n* transferred are received, the file can be reconstructed.

C. ONE HOP EXPERIMENT

The general arrangement for the one hop scenario is shown in Figure 12.

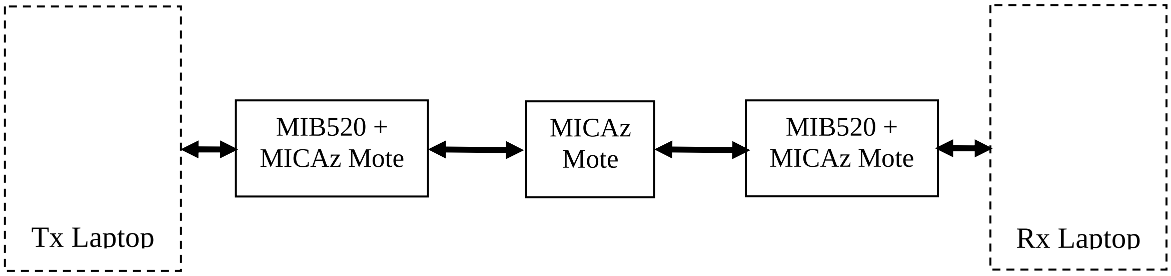


Figure 12. Scenario 2 - Terminal to Terminal via One Hop

To facilitate adding a hop between the two terminals, the power of the transmitting mote was reduced to a reference level two, which is believed to be equivalent to roughly -35 dBm. This equated to a transmission range of 1 ft. Due to the size limitation of the anechoic chamber, this short first hop would be necessary later for the two hop experiment. The power level for the mote used for the hop was set to -24 dBm.

Run	Non-Encoded				Non-Encoded (Redundant)				FEC Encoded			
	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)
1	299	238	0	N	598	288	214	N	640	501	0	Y
2	299	239	0	N	598	288	204	N	640	452	0	Y
3	299	229	0	N	598	296	235	N	640	451	0	Y
4	299	204	0	N	598	279	168	N	640	475	0	Y
5	299	233	0	N	598	283	178	N	640	472	0	Y
6	299	243	0	N	598	266	118	N	640	485	0	Y
7	299	227	0	N	598	260	12	N	640	446	0	Y
8	299	205	0	N	598	260	19	N	640	452	0	Y
9	299	230	0	N	598	248	127	N	640	453	0	Y
10	299	230	0	N	598	277	149	N	640	456	0	Y
11	299	219	0	N	598	282	165	N	640	500	0	Y
12	299	208	0	N	598	270	161	N	640	465	0	Y
13	299	221	0	N	598	256	147	N	640	503	0	Y
14	299	228	0	N	598	256	131	N	640	503	0	Y
15	299	221	0	N	598	257	148	N	640	505	0	Y
16	299	212	0	N	598	227	99	N	640	509	0	Y
17	299	230	0	N	598	280	144	N	640	471	0	Y
18	299	219	0	N	598	264	139	N	640	488	0	Y
19	299	219	0	N	598	274	156	N	640	500	0	Y
20	299	210	0	N	598	256	139	N	640	485	0	Y
21	299	236	0	N	598	270	134	N	640	523	0	Y
22	299	212	0	N	598	255	113	N	640	444	0	Y
23	299	230	0	N	598	276	146	N	640	446	0	Y
24	299	228	0	N	598	263	144	N	640	473	0	Y
25	299	234	0	N	598	254	136	N	640	475	0	Y

Table 3. One Hop JPEG Transmission Results

Once again the layout of the motes was selected to force dropped packets to occur. The results in Table 3 show that both of the non-encoded tests yielded no successful transmissions while the FEC encoded test was successful 100% of the time. Without redundancy, an average of 224 of the 299 necessary packets was received. Doubling the number of packets increased the average to 267 with an average of 141 redundant packets, although no successful transmissions were obtained. The FEC approach yielded an average of 459 of the 640 encoded packets received.

D. TWO HOP EXPERIMENT

The last of the scenarios, the two hop arrangement, is pictured in Figure 13.

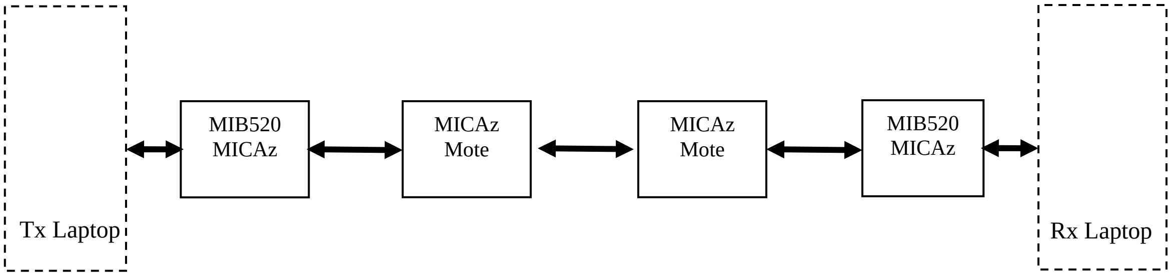


Figure 13. Scenario 3 - Terminal to Terminal via Two Hops

The arrangement of the terminal mote and first hop mote remain unchanged from the previous one hop test. The second hop mote, like the first, was set to -24 dBm for transmit power and located roughly 100 inches from the first mote. The arrangement of motes was set in order to make sure the coverage for each mote was as shown in Figure 14.

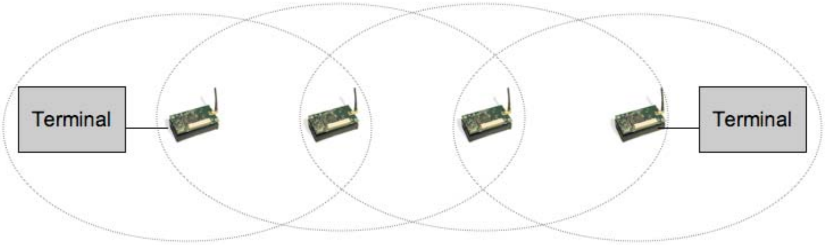


Figure 14. Desired Mote Coverage For Two Hop Test

Run	Non-Encoded				Non-Encoded (Redundant)				FEC Encoded			
	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)	No Packets Transmitted	No Packets Received	Redundant Packets	Success (Y/N)
1	299	151	279	N	598	239	545	N	640	458	110	Y
2	299	194	182	N	598	271	244	N	640	418	103	Y
3	299	186	18	N	598	282	306	N	640	405	314	Y
4	299	196	118	N	598	272	482	N	640	453	221	Y
5	299	198	108	N	598	264	533	N	640	398	412	Y
6	299	136	263	N	598	267	544	N	640	408	399	N
7	299	130	286	N	598	295	455	N	640	231	2	N
8	299	175	16	N	598	292	285	N	640	463	172	Y
9	299	201	109	N	598	157	455	N	640	156	56	N
10	299	174	108	N	598	285	302	N	640	408	346	Y
11	299	134	114	N	598	268	485	N	640	302	156	N
12	299	137	34	N	598	255	502	N	640	344	122	N
13	299	156	156	N	598	249	433	N	640	415	325	Y
14	299	165	167	N	598	289	526	N	640	285	208	N
15	299	165	185	N	598	299	285	Y	640	455	265	Y
16	299	174	221	N	598	297	465	N	640	468	214	Y
17	299	189	45	N	598	285	475	N	640	452	232	Y
18	299	198	49	N	598	246	388	N	640	385	126	N
19	299	156	145	N	598	264	475	N	640	408	188	N
20	299	164	164	N	598	255	385	N	640	415	182	N
21	299	134	185	N	598	269	352	N	640	375	55	N
22	299	137	235	N	598	285	387	N	640	435	164	Y
23	299	199	141	N	598	287	402	N	640	387	198	N
24	299	134	246	N	598	288	405	N	640	452	235	Y
25	299	155	270	N	598	292	265	N	640	407	185	N

Table 4. Two Hop JPEG Transmission Results

The results of the two hop tests are shown in Table 4. Once again the FEC encoded data tests outperformed the non-encoded data tests. Adding redundancy to the non-encoded tests raised the success rate from 0% to 4%, or an average number of packets received from 166 to 270. The average redundant packets increased from 154 to 415. By using the FEC encoding, the success rate was increased to 52%. An average of 403 packets out of the 640 sent was received with an average of 200 redundant packets.

The two hop arrangement added a large number of redundant packets to the link and caused a considerable reduction of link performance as a result. The increase of repeated packets seemed to cause a large number of necessary packets to be dropped during transmission and even the encoded tests showed difficulty in transferring complete JPEG images.

E. FEC PERFORMANCE OVER VARYING DISTANCES

The previous scenarios that were run were setup to purposefully cause a considerable amount of packet loss in order to test the FEC effectiveness. Each of the transceivers were positioned at a certain location and set to a power level that would force this condition. While the tests did show that the FEC scheme provided improved performance over the other tests run, another series of tests was needed in order to give a more qualitative comparison of the three schemes.

Distance (in)	No of Trials	Non-Encoded			Non-Encoded (Redundant)			FEC Encoded		
		Avg Packets Received	Avg Red Packets	% Success	Avg Packets Received	Avg Red Packets	% Success	Avg Packets Received	Avg Red Packets	% Success
43	15	299	0	100						
49	15	298.8	0	80						
55	15	298.3	0	67						
61	15	298.6	0	73						
67	15	298.4	0	73						
73	15	298.1	0	67	299	299	100	640	0	100
79	15	298.5	0	80	298.9	297.5	100	638.7	0	100
85	15	298.2	0	67	299	297	100	638.4	0	100
91	15	297	0	33	298.9	296.8	87	635.5	0	100
97	15	295.6	0	0	298.9	296	93	635.2	0	100
103	15	288.1	0	0	298.9	292.6	87	615.5	0	100
109	15	220	0	0	298.1	275.7	27	473.9	0	87
115	15				292.1	228.9	0	398.3	0	53
121	15				272.2	173.4	0	282.6	0	7
127	15				287.7	211.5	0	370.7	0	60
133	15				75	12	0	78	0	0
139	15									

Table 5. Effect of Varying Distances Upon Transmission Success

In order to get a better idea of how the performance was falling off for each of the transmission methods as the distance increased, the final test was performed. The configuration of hardware used for this test was that of the original terminal to terminal test. The transmitting terminal, set to a power level of -24 dBm, remained fixed while the receiving terminal was varied to record the packet transmission effects. Each of the non-encoded and encoded methods were tested fifteen times at varied distances to find the transition from 100% image transfer success to the distance that resulted in a failure to transfer a single image.

Table 5 displays the results from the test. The non-encoded scheme showed a gradual decrease in link performance starting at a distance of 43 inches until it totally failed at 97 inches. At that distance, the non-encoded scheme that included redundant packets was still showing a 93% success rate. At 115 inches, the redundant packet scheme completely failed. At this distance the FEC encoded scheme still performs at a 53% success rate. Not till 133 inches did it fail to transfer a single image. There was an unexpected spike in performance for the FEC scheme at the next to last distance that was unexplained.

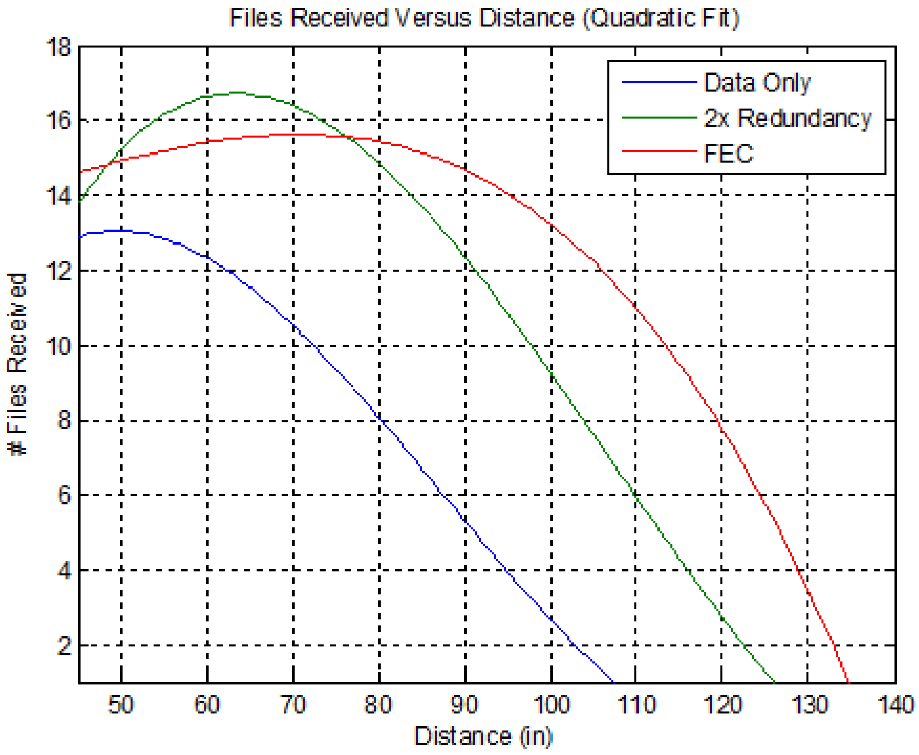


Figure 15. Quadratic Fit Curves for Experimental Results

In order to get a better picture of the performance falloff, the plot shown in Figure 15 was created from the recorded data. To achieve the smooth curves shown, quadratic fit curves were created from the data. To compare the three transmission schemes, a reference for acceptable loss of 25% was chosen. This translated to receiving 11.25 out of the 15 files sent. By that standard the non-redundant scheme that included redundancy represented a 42% increase in transmitting range from the non-redundant scheme. The FEC encoded scheme represented a 66% increase in transmitting range.

F. SUMMARY

For the experiment, results were taken for three different configurations in order to demonstrate the effectiveness of the FEC algorithm under differing conditions. In each of the configurations tested, the FEC encoded transmissions achieved better results at greater ranges than the non-encoded tests. The erasure coding allowed for significant packet loss to occur while still recovering the image at the receiving terminal. While the redundant example showed some improvement over the non-redundant test, consecutive lost packets quickly became an issue as the receiver’s distance from the transmitter was increased.

Both the terminal to terminal and one hop configurations yielded very similar results. The two hop configuration was similar in that it showed the FEC test to be the most effective, but it also highlighted some performance limitations of the medium access protocol used by the lab motes that should be investigated further. The packet losses at the receiving node were much greater with the two hop configuration. It appeared that the increased network congestion was responsible for the reduced performance.

V. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

For each of the experiment configurations used, the erasure code outperformed the non-FEC schemes. The larger the amount of data to be sent, the more effective erasure code would be for the transmission process. While the TinyOS AM packet structure is very limiting, the positive effects of the coding scheme were still observed.

While the error correction method did improve the performance of the channel, the two hop experiment pointed out flaws with the underlying architecture that must be addressed. During the two hop experiment, a large amount of packet loss was observed that can be attributed to unnecessary congestion in the network. The CSMA scheme used by the motes may be the reason for the packet loss observed.

Maybe the most limiting factor observed was the channel throughput. During the terminal to terminal tests an average transfer time of 7 sec was recorded for the non-redundant file transfer. This equated to a transfer rate of 12.3 kbps for the 299 packets. Considering that the CC2420 radio on the MICAz mote was rated for 250 kbps and the serial forwarder program was set for a transfer rate of 57.6 kbps, the cause of the slow transmission speed was unknown. Researching the TinyOS documentation led to the discovery that the MAC protocol was actually the limiting factor. The CSMA protocol limited the number of packets sent to the radio each second to 43. Implementing a TCP/IP bridge to pass multimedia or VOIP traffic would require a much higher transmission rate. While the equipment and software used in this experiment might not be able to achieve the desired results, other MAC schemes may be available that could provide a transmission rate closer to the radio's capabilities.

B. RECOMMENDATIONS

The first recommendation would be to conduct further tests on the erasure code to find what parameters of *k* and *n* would yield the most optimum coding scheme. During the course of the experiment, these values were only adjusted to manipulate the number of blocks that would be encoded. A more in-depth study to find out if these values can be more useful to the overall channel performance might be beneficial.

Erasure code did provide very positive results during the experiment. A positive next step would be take the performance of the erasure code and compare it to a convolutional or turbo code. Convolutional codes and turbo codes represent some of the more advanced and efficient codes available today.

Due to the limitations provided by the CSMA MAC scheme installed on the experimental motes, work involving newer MAC schemes may provide better results. With TinyOS 1.1.3, the B-MAC MAC scheme was introduced. B-MAC increased the rate at which packets are sent to the radio slightly over the CSMA scheme used during this experiment. Improvements to this transmission rate above that provided by B-MAC may also be available in versions later than 1.1.3.

Finally, an ideal sensor network would most likely combine some form of forward error correction with a limited implementation of ARQ. In order to make this possible, it may be necessary to combine the ARQ with a suitable MAC scheme as mentioned before. The Z-MAC scheme introduced earlier might be a good fit with its congestion dependent combination of CSMA and TDMA.

APPENDIX

A. SNAIL CLIENT CODE

```
// SNAILClient_fec.java
//
// Java program prompts user for selection and collects data ...
// to be transferred to SNAILServer
//
// Last Updated by Thomas Childers  26Nov08

package net.tinyos.tools1;

import com.sun.image.codec.jpeg.ImageFormatException;
import com.sun.image.codec.jpeg.JPEGCodec;
import com.sun.image.codec.jpeg.JPEGImageEncoder;
import javax.swing.*;
import java.awt.image.BufferedImage;
import java.net.*;
import java.io.*;
import javax.imageio.ImageIO;

public class SNAILClient_fec {

    private static byte[] concatenate(byte[] a, byte[] b)
    {
        if (a == null) {
            return b;
        } else {
            byte[] bytes = new byte[a.length + b.length];

            System.arraycopy(a, 0, bytes, 0, a.length);
            System.arraycopy(b, 0, bytes, a.length, b.length);
            return bytes;
        }
    }

    /** Creates a new instance of TcpClient */
    public SNAILClient_fec() {
    }

    public  static  byte[]  bufferedImageToByteArray(BufferedImage  img)
throws ImageFormatException, IOException
    {
        ByteArrayOutputStream os = new ByteArrayOutputStream();
        JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(os);
        encoder.encode(img);
        return os.toByteArray();          }


    // ------------------------------------------------------------------
    public static void main( String args[] )
    {
```

```
Socket clientSocket;
  String DIGEST_ALGORITHM = "sha";
  byte[] outy = null;

try
  {

     while(true)
     {
           // Initialization for File Transfer
           FileInputStream inputFile = null;
             FileOutputStream outputFile = null;
             BufferedReader input = null;
             BufferedWriter output = null;
           char NewArray[] = null;
                 String FinalFileData = null;

           // This is the portion that attempts to collect data
           from the console
           System.out.print("\nActions available
           currently:\n\n\t (1) - Send Instant Msg \n\t (2) -
           Standard File Transfer \n\t (3) - Jpeg Transfer");
           System.out.print("\n\n Enter Option Number: ");
             BufferedReader instruction = new BufferedReader
           (new InputStreamReader(System.in));
           String processInstruction = instruction.readLine();
           int instrObtained =
           Integer.parseInt(processInstruction);
           String FinalOutputData = new String();

           // This asks the user whether or not to encode the
           data
                 System.out.print("\nFEC Encoding:\n\n\t (1) -
           Encode Data \n\t (2) - Data Tx Only \n\t");
           System.out.print("\n\n Enter Coding Selection: ");
             BufferedReader encoding = new BufferedReader (new
           InputStreamReader(System.in));
           String encodingoption = encoding.readLine();
           int encodingchoice =
           Integer.parseInt(encodingoption);
                 byte encoded = 0;
                 if (encodingchoice == 1)
                     encoded = 0;
                 if (encodingchoice == 2)
                     encoded = 1;

           // ----------------------------------------------
           // Next section does based on the option selected

           if(instrObtained == 1) // Loop for short message
           services
           {
                 System.out.print("Enter message that needs to
                 be transmitted: ");
```

```
BufferedReader inStream = new BufferedReader
(new InputStreamReader(System.in));
String outputData = inStream.readLine();

// The following is to add in the flag for
option selected
String dataReceived = outputData;
// Conversion of string to char array
char dataArray [] = dataReceived.toCharArray();

NewArray = new char[ dataArray.length ];
// Additional 2 segments in array for
optionSelect and Filesize
for(int i = 0; i < (dataArray.length); i++)
{
      NewArray[i] = dataArray[i];
}

byte[] codeArray = new byte[3];
codeArray[0] = '1';     // Flag to server that
this is a short message service
codeArray[1] = '0';
codeArray[2] = encoded;

// This routine is to check if the mapping of
array is correct. Uncomment when required to
use.
// for (int count = 0; count < NewArray.length;
count++)
//    {
//     System.out.print(" " +NewArray[count]);
//    }

for( int count = 0; count < NewArray.length;
count++)
{
       FinalOutputData += NewArray[count];
}
// End of adding option selection flag

byte[] foutdata = FinalOutputData.getBytes(
"8859_1" /* encoding */ );  // Putting in byte
array form for transfer

outy = concatenate(codeArray,foutdata);
}

else if(instrObtained == 2) // This is the loop
for File transfer
{

JFileChooser fileChooser = new
JFileChooser(".");
int status = fileChooser.showOpenDialog(null);
String filename = new String();
```

```
      if (status == JFileChooser.APPROVE_OPTION)
      {
            File            selectedFile             =
            fileChooser.getSelectedFile();
            System.out.println("Selected:      "      +
            selectedFile.getPath());
            filename = selectedFile.getPath();
      }

      File fileIn;
      BufferedInputStream in = null;
      BufferedOutputStream out = null;
      byte[] filedata = null;

      try
      {
            fileIn = new File(filename);
            System.out.println("File   size   is   "   +
            fileIn.length() + " bytes");
            in      =   new   BufferedInputStream(new
            FileInputStream(fileIn));

            byte[]       fileContent       =       new
            byte[(int)fileIn.length()];
            in.read(fileContent);
            filedata = fileContent;
      }
      catch (IOException ex)
      {
            ex.printStackTrace();
      }
      finally
      {            // always close the streams
            try
            {
                if (in != null) in.close();
                if (out != null) out.close();
            }
            catch (IOException ex)
            {
                   ex.printStackTrace(); }
            }

      //   Putting  the  options  with  the  data  into
      FinalOutputData
      byte[] option = new byte[3];
      option[0] = (byte)'2';
      option[1] = (byte)'0';
      option[2] = encoded;

      outy = concatenate(option,filedata);

}
// End of text file transfer portion
```

```
else if(instrObtained == 3) // Jpeg Transfer Portion
{
      JFileChooser fileChooser = new
      JFileChooser(".");
      int status = fileChooser.showOpenDialog(null);
      String filename = new String();
      if (status == JFileChooser.APPROVE_OPTION)
      {
            File selectedFile =
            fileChooser.getSelectedFile();
            System.out.println("Selected: " +
            selectedFile.getPath());
            filename = selectedFile.getPath();
      }

      BufferedImage img = null;
      try
      {
            img = ImageIO.read(new File(filename));
      }
      catch (IOException e) {}

      ByteArrayOutputStream baos=new
      ByteArrayOutputStream();
      JPEGImageEncoder encoder
      =JPEGCodec.createJPEGEncoder(baos);
      encoder.encode(img);
      byte [] fileData = baos.toByteArray();
      //System.out.println("File size is " +
      fileData.length() + " bytes"); !! Find
      length of byte array???

      // Putting the options with the data into
      FinalOutputData
      byte[] option = new byte[3];
      option[0] = (byte)'3';
      option[1] = (byte)'0';
      option[2] = encoded;

      // Now need to combine image data and option
      data

      outy = concatenate(option,fileData);

}

// Below will send the collected data out to
SNAILserver
InetAddress srcAddress = InetAddress.getLocalHost();

int srcListeningPort = 4567;
clientSocket = new Socket( srcAddress,
srcListeningPort );
```

```
                  System.out.println("Client:          Attempting     to
                  established connection " );

                  //int  len  =  outy.length;  //  If  I  were  to  find  the
                  length of the byte array

//                InputStream is = clientSocket.getInputStream();
//                DataInputStream dis = new DataInputStream( is );
//
                  OutputStream os = clientSocket.getOutputStream();
//                DataOutputStream dos = new DataOutputStream( os );

//                dos.write( outy, 0, outy.length );
                  ObjectOutputStream oos = new ObjectOutputStream(os);

                  // get the socket output stream
                  oos.writeObject(outy);

                  System.out.println( "\n\n");
                  System.out.println("Information transmitted");
                  if (instrObtained == 1)
                  {
                        String  t  =  new  String(  outy  ,  "Cp1252"  /*
                        encoding */ );
                        System.out.println( t );
                  }
                  System.out.println( "\n\n");

            }       // End of while-loop
       }      //  End of Try Statement
         catch( Exception e )
         {
                System.out.println( e );
         }
    }  // End Main Method

}  //  End SNAILClient_fec
```

B. SNAIL SERVER CODE

```
// SNAILserverTest3_fec.java
//
// Java program encodes the data if required and packages the data ...
// for transfer over the sensor network
//
// Last Updated by Thomas Childers  26Nov08

package net.tinyos.tools;

import java.io.*;
import java.net.*;
import java.util.Properties;
import net.tinyos.message.*;
import net.tinyos.util.*;

//  Added for FEC Operations
import com.onionnetworks.fec.FECCode;
import com.onionnetworks.fec.FECCodeFactory;
import com.onionnetworks.util.Buffer;

public class SNAILserverTest3_fec
{

    public static final short TOS_BCAST_ADDR = (short) 0xffff;
    static Properties p = new Properties();

    /** Creates a new instance of TcpServer */
    public SNAILserverTest3_fec()
    {}

    public static byte restoreSequenceNo()      //  This  instance  is  to
    obtain the sequence number of the packet
    {
      try
      {
            FileInputStream             fis             =             new
            FileInputStream("bcast.properties");
            p.load(fis);
            byte  i  =  (byte)Integer.parseInt(p.getProperty("sequenceNo",
            "1"));
            fis.close();
            return i;
      }

      catch (IOException e)
      {
              p.setProperty("sequenceNo", "1");
              return 1;
      }
    }

    public static void saveSequenceNo(int i)    //  This  is  an  instance
    to save the sequence number
    {
      try
      {
            FileOutputStream            fos            =            new
            FileOutputStream("bcast.properties");
            p.setProperty("sequenceNo", Integer.toString(i));
            p.store(fos, "#Properties for BcastInject\n");
      }
      catch (IOException e)
      {
            System.err.println("Exception while saving sequence number"
            + e);
          e.printStackTrace();
      }
    }
```

```
public static void main( String args[] )    // This is the main
instance where the program runs
{
  byte sequenceNo = 0;
  boolean read_log = false;

  ServerSocket serverSocket;
  try
  {
        // without specifying on the interface
        // it will listens to all interfaces; wildcard
        serverSocket = new ServerSocket( 4567 );

        while( true )                   //  Infinite Loop
        {
              System.out.println("  ");
              System.out.println(" ----------------Start  Of  Server
              Service ---------------" );
              System.out.println(" Server:   No  Client  Yet;  Please
              Wait Longer ...");
              Socket aSocket = serverSocket.accept();
              System.out.println(" Server:  Client arrived!");

              InputStream is = aSocket.getInputStream();
              // reading in byte array
              ObjectInputStream ois = new ObjectInputStream(is);
              byte[] data = (byte[])ois.readObject();
              int arraysize = data.length;
              char data1 = (char) data[0];     // Pull out options
              selected
              char data2 = (char) data[1];
              //char data3 = (char) data[2];
              byte encoded = data[2];

              System.out.println(" ");
              //  -----------------------------------------------
              //  Data has been read in, now to Encode and Transmit

              SNAILmsg_tc packet = new SNAILmsg_tc();

              packet.set_hop_count((short)0);
              packet.set_source(0);

              try
              {
                 int optionSelect = 0;               //         This
                 variable is in ascii due to input from terminal
                 int FileNameSize = 0;               //         This
                 variable will be in integer value
                 int count = 0;
                 int j = 0;                    //      Variable   to
                 store  characters converted to Hex
                 int end = 1;                  //  Counter  for  next
                 packet to be sent
```

```
int start = 0;                     //   Counter   for
number of packets sent
int z = 0;                         // Loop counter
int test = 0;                      //   Counter   for
character count
int packetcount = 0;               // Packet Counter
int newIndexStart = 0;
int newIndexEnd = 0;
int oldarray = 0;
int bytesCopied = 0;                // Running counter
of the number of bytes sent out
int padding = 0;
byte TerminalID = '2';             //   This   ID   is
unique to every terminal. Not really utilized.
int TxScanFlag = '0';                 // Old variable.
Not used.
long delay = 5;                    //     Delay     in
between packets transmission
long delay2 = 160;                       // Delay for
handshake packet transmission

byte queueArray [] = new byte[29];
byte FinalDataArray [] = null;
byte headerArray [] = new byte[29];



System.out.println("Total   Characters   Received:   "
+arraysize);
System.out.print("\n");

System.out.print("\nOption Selected: " + data1);
System.out.print("\nFile Name Size : " + data2);
System.out.println("\n");
System.out.print("\n\n");

optionSelect = data1;

System.out.print("\n");


// ---------------------------------------------------
// Finding Out what kind of packet structure we are
dealing with

arraysize       =       arraysize       -       3;
// Not transmitting the option 3 digit code
int   k2   =   (arraysize/(packet.dataLength()-3)+1);
// # packets before encoding
if  ((arraysize  %  (packet.dataLength()-3))  ==  0)
// Fix case if evenly divisible
k2 = k2 - 1;
int       n2       =       2       *       k2;
// n is # of encoded packets
```

```
int packetsize = 26;
// AM message determined data size (29) minus 3 bytes

int k = 64;
// Number of packets to encode
int n = 128;
// Number of encoded packets
if (data1 == '1')
// If this is a text message or non-encoded, i want
to fix k and n by msg length
{
     k = k2;
     n = n2;
}

if (encoded == 1)
{
      k = (arraysize/(packet.dataLength()-2)+1);
      // # packets before encoding
      if ((arraysize % (packet.dataLength()-2)) == 0)
      // Fix case if e  venly divisible
      k = k - 1;
}


System.out.print("\nArray Size : " +arraysize);
int totalBits = k*packetsize;

//System.out.println("\nTotal bytes : " +totalBits);
//padding = totalBits - (arraysize - 3);
// Due to 1 additional bit per payload
//System.out.println("\nCharacters Sent: "
+arraysize);
//System.out.println("\nPackets required: " +
packetcount);



// ---------------------------------------------
// Fix the format of the arraysize and the
packetcount so they can be transmitted in 2 bytes
each

byte[] arraysz = new byte[]        // Convert
arraysize to 2 bytes
{
    (byte)(arraysize >>> 16),
    (byte)(arraysize >>> 8),
    (byte) arraysize
};

byte[] ksz = new byte[]        // Convert k to 2
bytes
{
    (byte)(k >>> 8),
     (byte) k
};

byte[] nsz =  new byte[]        //  Convert n to 2
bytes
      {
         (byte)(n >>> 8),
         (byte) n
      };


      // -----------------------------------------
      //  Find out out many image blocks there will
      be (can't encode entire image at once *slow*)

      int block = k * packetsize;
      // Size of image block
      int numblocks = (arraysize / block) + 1;
      // Number of image blocks to encode
      if ((arraysize % block) == 0)
      // Fix case if evenly divisible
      numblocks = numblocks - 1;

      //  -----------------------------------------
      //  Need to store each of the image data blocks

      byte[] source = new byte[arraysize];
      // Contains all of data minus the 3 byte option
      System.arraycopy(data, 3, source, 0,
      arraysize);

      byte[][] source2 = new byte[numblocks][block];

      if (numblocks == 1)
      // If less than blocksize of data
      {
            for (int i=0; i < arraysize; i++)
            source2[0][i] = source[i];
      }
      else
      {
            for (int i=0; i < (numblocks-1); i++)
            //  Puts data blocks in correct place
            System.arraycopy(source, i*block ,
            source2[i], 0, block);
            for (int i=0; i < (arraysize % block);
            i++)
            source2[numblocks-1][i] =
            source[((numblocks-1)*block) + i];
      }


      //  ------------------------------------------
      //  FEC Setup Procedure  //
```

```
byte[][] repair = new
byte[numblocks][n*packetsize]; //this is
our encoded data
int[][] repairIndex = new int[numblocks][n];

//These buffers allow us to put our data in
them they
//reference a packet length of the file (or at
least will once
//we fill them)

//create our fec code
if (encoded == 0)       // I only want to do
this section if we are going to encode
{
      FECCode fec =
      FECCodeFactory.getDefault().createFECCode
      (k, n);  // creating code

      for (int w=0; w < numblocks; w++)
      {
         Buffer[] sourceBuffer = new Buffer[k];
         Buffer[] repairBuffer = new Buffer[n];

         for (int i = 0; i <
         sourceBuffer.length; i++)
         sourceBuffer[i] = new
         Buffer(source2[w], i*packetsize,
         packetsize);

         for (int i = 0; i <
         repairBuffer.length; i++)
         repairBuffer[i] = new
         Buffer(repair[w], i*packetsize,
         packetsize);

         //When sending the data you must
         identify what it's index was.
         //Will be shown and explained later

         for (int i = 0; i <
            repairIndex[w].length; i++)
            repairIndex[w][i] = i;


         //encode the data
         fec.encode(sourceBuffer, repairBuffer,
         repairIndex[w]);
      }
}
//  End FEC Encoding //
//  -------------------------------------

if (encoded == 0)
{
```

```
//  This routine is to send out the handshake
packet. To inform receiver, how many
//  packets to expect and bytes of padding
required.  Also sent, the repair packet index

sequenceNo = restoreSequenceNo();
sequenceNo++;
packet.set_seqno(sequenceNo);
System.out.print("\nThis is the headerpacket
sequence number: " +sequenceNo);
saveSequenceNo(sequenceNo);

System.out.println("\nSending Header Packet");
headerArray [0] = (byte) sequenceNo;       //
Sequence number
headerArray [1] = (byte) TerminalID;       //
Second byte contains the Terminal ID
headerArray [2] = (byte) optionSelect;     //
Third byte contains the option selected
headerArray [3] = (byte) FileNameSize;     //
Fourth byte contains the FileName length  (not
used)
headerArray [4] = encoded;                 //
Fifth byte contains Encoding Option
headerArray [5] = ksz[0];                  //
6th/7th byte contains ksz info (1)
headerArray [6] = ksz[1];                  //
(2)
headerArray [7] = nsz[0];                  //
8th/9th byte contains nsz info (1)
headerArray [8] = nsz[1];                  //
(2)
headerArray [9] = arraysz[0];              //
10th/11th/12th byte contains arraysize info (1)
headerArray [10] = arraysz[1];             //
(2)
headerArray [11] = arraysz[2];             //
(3)
headerArray [12] = (byte) 69;              //
Handshake Packet Queue (not used)
MoteIF mote = new
MoteIF(PrintStreamMessenger.err);

for(int redundancy = 4; redundancy > 0;
redundancy--)
{

      for ( int i = 13; i < 29 ; i++)    //
      Clear the rest of the payload
      {
          headerArray [i] = 00;
      }
```

```
        for( int i = 0; i < headerArray.length;
        i++)
        System.out.print(" " + (headerArray[i]));

        for (int i = 0; i < packet.dataLength();
        i++) // This boundary is fixed due to 29
        bytes limitation by AM
        {
            int c = headerArray[i];
            packet.set_action(i,c);
            // Contents in packet is now
            Hexadecimal

            packet.set_seqno(sequenceNo);
            // i points to the start of the array
            count = i+1;
            // Characters counter
        }

        System.out.println("\n");


        Thread.sleep( delay2 );
        mote.send(TOS_BCAST_ADDR, packet);
        Thread.sleep( delay2 );
}
//  End of handshake packet transmission
//  -----------------------------------------

//  Now transmitting data packets

for    (int    y=0;    y    <    numblocks;    y++)
// Out loop for transmitting all of the data
blocks
{
    packetcount = n;
    int        TotalPacketCount        =        0;
    // Will be used to store packet #
    bytesCopied = 0;
    System.out.println("Sending block: " +
    (y+1)+" of " +numblocks);
    for ( ; packetcount > 0; packetcount--)
    {
       sequenceNo = restoreSequenceNo();
       sequenceNo++;
       packet.set_seqno(sequenceNo);
       //System.out.print("\nThis is the payload
       sequence number :" +sequenceNo);

       //System.out.println("\nSending Packet
       Number :" + (TotalPacketCount+1));

       saveSequenceNo(sequenceNo);
```

```
queueArray[0]       =       (byte)       y;
// Storing Block number to be sent


byte[] countsz =  new byte[]        //
Convert n to 2 bytes
{
    (byte)(TotalPacketCount >>> 8),
    (byte) TotalPacketCount
};

queueArray[1]    =    (byte)    countsz[0];
// Storing 2 byte Packet Number
queueArray[2] = (byte) countsz[1];


for ( int counter = 3; (counter < 29) ;
counter++)
{
   queueArray [counter] =
   repair[y][bytesCopied];    // Conversion
   to byte array of Data
   bytesCopied++;

   //System.out.println(counter+" "+y+"
   "+bytesCopied+" "+);
}
//bytesCopied--;

// -------------------------------------
// Send Packets

for (int redundancy = 1; redundancy > 0;
redundancy--)       // Sending Packets
{
for (int i = 0; i < packet.dataLength();
i++)
{
//byte c = queueArray[i];
//j = (int) c;
// j contains the decimal integer

packet.set_action(i,queueArray[i]);
// Contents in packet is now Hexadecimal

//packet.set_seqno(sequenceNo);
// i points to the start of the array

// j is the contents of the array

//System.out.print((packet.dataGet()[i])+
" ");
count = i+1;                          //
Characters counter
                  }
```

```
             Thread.sleep( delay );

             mote.send(TOS_BCAST_ADDR, packet);
          }  // End transmission loop

          saveSequenceNo(sequenceNo);
          TotalPacketCount++;
     }  //  End inner for loop
}  //  End Block Tx for loop

// -----------------------------------------------
// Now transmitting a termination packet

sequenceNo = restoreSequenceNo();
sequenceNo++;
packet.set_seqno(sequenceNo);
System.out.print("\nThis  is  the  termpacket  sequence
number: " +sequenceNo);
saveSequenceNo(sequenceNo);
byte[] termArray = new byte[29];

System.out.println("\nSending Terminating Packet");
termArray [1] = (byte) 69;         // Flags
termArray [2] = (byte) 69;         //
termArray [3] = (byte) 69;         //


for(int redundancy = 10; redundancy > 0; redundancy--)
{

    for ( int i = 4; i < 29 ; i++)    // Clear the rest
    of the payload
    {
        termArray [i] = 00;
    }

    for( int i = 0; i < termArray.length; i++)
        System.out.print(" " + (termArray[i]));

    for (int i = 0; i < packet.dataLength(); i++) //
    This boundary is fixed due to 29 bytes limitation
    by AM
    {
       int c = termArray[i];
       packet.set_action(i,c);                   //
       Contents in packet is now Hexadecimal

       packet.set_seqno(sequenceNo);             //
       i points to the start of the array
       count = i+1;                              //
       Characters counter
                    }

       System.out.println("\n");
       Thread.sleep( delay );
       mote.send(TOS_BCAST_ADDR, packet);
       Thread.sleep( delay );

    }   // end for loop


} // end if (encoded)

//  -----------------------------------------------
//  -----------------------------------------------

else if (encoded == 1)  // if non-encoded
{
   //  This routine is to send out the handshake
   packet. To inform receiver, how many
   //  packets to expect and bytes of padding required.
   Also sent, the repair packet index

   sequenceNo = restoreSequenceNo();
   sequenceNo++;
   packet.set_seqno(sequenceNo);
   System.out.print("\nThis is the headerpacket
   sequence number: " +sequenceNo);
   saveSequenceNo(sequenceNo);

   System.out.println("\nSending Header Packet");
   headerArray [0] = (byte) sequenceNo;       //
   Sequence number
   headerArray [1] = (byte) TerminalID;        // Second
   byte contains the Terminal ID
   headerArray [2] = (byte) optionSelect;      // Third
   byte contains the option selected
   headerArray [3] = (byte) FileNameSize;      // Fourth
   byte contains the FileName length  (not used)
   headerArray [4] = encoded;                  //  Fifth
   byte contains Encoding Option
   headerArray [5] = ksz[0];                  //
   6th/7th byte contains ksz info (1)
   headerArray [6] = ksz[1];                  // (2)
   headerArray [7] = nsz[0];                  //
   8th/9th byte contains nsz info (1)
   headerArray [8] = nsz[1];                  // (2)
   headerArray [9] = arraysz[0];              //
   10th/11th/12th byte contains arraysize info (1)
   headerArray [10] = arraysz[1];             // (2)
   headerArray [11] = arraysz[2];             // (3)
   headerArray [12] = (byte) 69;              //
   Handshake Packet Queue (not used)
   MoteIF mote = new MoteIF(PrintStreamMessenger.err);

   for(int redundancy = 4; redundancy > 0; redundancy--
   )
   {
```

```
   for   (   int   i   =   13;   i   <   29   ;   i++)
   // Clear the rest of the payload
   {
      headerArray [i] = 00;
   }

   for( int i = 0; i < headerArray.length; i++)
     System.out.print(" " + (headerArray[i]));

   for  (int  i  =  0;  i  <  packet.dataLength();  i++)
   // This boundary is fixed due to 29 bytes
   limitation by AM
   {
     int c = headerArray[i];
     packet.set_action(i,c);                     //
     Contents in packet is now Hexadecimal

     packet.set_seqno(sequenceNo);         //
     i points to the start of the array
     count = i+1;                    // Characters
     counter
   }

   System.out.println("\n");
   Thread.sleep( delay );
   mote.send(TOS_BCAST_ADDR, packet);
   Thread.sleep( delay );
}

//  End of handshake packet transmission
//  --------------------------------------------

//  Now transmitting data packets


packetcount = k;
int TotalPacketCount = 0;
// Will be used to store packet #
bytesCopied = 0;
for ( ; packetcount > 0; packetcount--)
{
   sequenceNo = restoreSequenceNo();
   sequenceNo++;
   packet.set_seqno(sequenceNo);
   //System.out.print("\nThis is the payload sequence
   number :" +sequenceNo);

   //System.out.println("\nSending Packet Number :" +
  (TotalPacketCount+1));

  saveSequenceNo(sequenceNo);
  //queueArray[1]      =      (byte)      TerminalID;
  // Storing Block number to be sent
```

```
   byte[]      countsz      =           new      byte[]
   // Converting packet number to two bytes
   {
      (byte)(TotalPacketCount >>> 8),
      (byte) TotalPacketCount
};

queueArray[0]        =        (byte)        countsz[0];
// Storing 2 byte Packet Number
queueArray[1] = (byte) countsz[1];

for ( int counter = 2; (counter < 29) && (bytesCopied
< arraysize) ; counter++)
{
   queueArray    [counter]    =    source[bytesCopied];
   // Conversion to byte array of Data
   bytesCopied++;

   //System.out.println(counter+" "+y+"
   "+bytesCopied+" "+);
}
//bytesCopied--;


// ------------------------------------------------------
// Send Packets

for (int redundancy = 1; redundancy > 0; redundancy--)
// Sending Packets
{
  for (int i = 0; i < packet.dataLength(); i++)
  {
     //byte c = queueArray[i];
     //j = (int) c;                              // j
     contains the decimal integer

     packet.set_action(i,queueArray[i]);
     // Contents in packet is now Hexadecimal

     //packet.set_seqno(sequenceNo);
     // i points to the start of the array

     // j is the contents of the array

     //System.out.print((packet.dataGet()[i])+ " ");
     count = i+1;                            //
     Characters counter
  }
  Thread.sleep( delay );
  //System.out.print("\n");
  //System.out.println("\n");
  mote.send(TOS_BCAST_ADDR, packet);
}  // End transmission loop
```

```
    saveSequenceNo(sequenceNo);
    TotalPacketCount++;
 }  //  End inner for loop


 // ------------------------------------------------------
 // Send terminating packet

 sequenceNo = restoreSequenceNo();
 sequenceNo++;
 packet.set_seqno(sequenceNo);
 System.out.print("\nThis is the termpacket sequence
 number: " +sequenceNo);
 saveSequenceNo(sequenceNo);
 byte[] termArray = new byte[29];

 System.out.println("\nSending Terminating Packet");
 termArray [1] = (byte) 69;         // Flags
 termArray [2] = (byte) 69;     //
 termArray [3] = (byte) 69;     //


 for(int redundancy = 5; redundancy > 0; redundancy--)
 {

   for ( int i = 4; i < 29 ; i++)    // Clear the rest of
   the payload
   {
      termArray [i] = 00;
   }

   for( int i = 0; i < termArray.length; i++)
      System.out.print(" " + (termArray[i]));

   for (int i = 0; i < packet.dataLength(); i++) // This
   boundary is fixed due to 29 bytes limitation by AM
   {
      int c = termArray[i];
      packet.set_action(i,c);               // Contents
      in packet is now Hexadecimal

      packet.set_seqno(sequenceNo);         //
      i points to the start of the array
      count = i+1;                             //
      Characters counter
   }

   System.out.println("\n");


   Thread.sleep( delay );
   mote.send(TOS_BCAST_ADDR, packet);
   Thread.sleep( delay );

}   // end for loop
```

```
         }  // end else for non-encoded option


       }  // end try statement
       catch(Exception e)
       {
            e.printStackTrace();
       }
       // --->


       System.out.println(" ----------------End Of Server Service -----
       ----------" );

       System.out.println("  ");




      }  // End infinite While Loop

     }  // End Main Class Try Statement
     catch( Exception e )
     {
          System.out.println( e );
     }
   }  // End Main Class

}  // End SNAILserverTest_fec
```

C. SNAIL LISTEN CODE

```
// SNAILlistenTest3_fec.java
//
// Java program takes in data from the sensor network and decodes if
// necessary.  Resulting data is displayed or stored as a file.
//
// Last Updated by Thomas Childers  26Nov08

// $Id: Listen.java,v 1.5 2004/08/19 00:13:49 idgay Exp $

/*
 *  "Copyright (c) 2000-2003 The Regents of the University of
California.
 * All rights reserved.
 *
 * Permission to use, copy, modify, and distribute this software and
its
 * documentation for any purpose, without fee, and without written
agreement is
 * hereby granted, provided that the above copyright notice, the
following
 * two paragraphs and the author appear in all copies of this software.
 *
 * IN NO EVENT SHALL THE UNIVERSITY OF CALIFORNIA BE LIABLE TO ANY
PARTY FOR
 * DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES
ARISING OUT
 * OF THE USE OF THIS SOFTWARE AND ITS DOCUMENTATION, EVEN IF THE
UNIVERSITY OF
 * CALIFORNIA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 * THE UNIVERSITY OF CALIFORNIA SPECIFICALLY DISCLAIMS ANY WARRANTIES,
 * INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.  THE SOFTWARE PROVIDED
HEREUNDER IS
 * ON AN "AS IS" BASIS, AND THE UNIVERSITY OF CALIFORNIA HAS NO
OBLIGATION TO
 * PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR
MODIFICATIONS."
 *
 * Copyright (c) 2002-2003 Intel Corporation
 * All rights reserved.
 *
 * This file is distributed under the terms in the attached INTEL-
LICENSE
 * file. If you do not find these files, copies can be found by writing
to
 * Intel Research Berkeley, 2150 Shattuck Avenue, Suite 1300, Berkeley,
CA,
 * 94704.  Attention:  Intel License Inquiry.
 */

import java.io.*;
import java.net.*;
import net.tinyos.util.*;
import net.tinyos.packet.*;
import java.util.*;
import javax.swing.*;

import com.onionnetworks.fec.FECCode;
import com.onionnetworks.fec.FECCodeFactory;
import com.onionnetworks.util.Buffer;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class SNAILlistenTest3_fec {

  public static void main(String args[]) throws IOException
  {
      if (args.length > 0)
      {
            System.err.println("usage: java net.tinyos.tools.Listen");
          System.exit(2);
      }

      Socket clientSocket;
```

```
PacketSource reader = BuildSource.makePacketSource();
if (reader == null) {
    System.err.println("Invalid packet source (check your MOTECOM
    environment variable)");
    System.exit(2);
}

BufferedWriter output = null;
FileOutputStream outputFile = null;

try
  {
     byte TerminalID = '3';
       reader.open(PrintStreamMessenger.err);
     byte sequenceNo = 0;
     byte [] HandshakePkt = new byte[34];
     byte [] buffer = new byte[34];

     while(true)
     {
       System.out.println("---------------------------------
       -------------------------------------------");
           System.out.println("Waiting ....");
       int j = 0;
       int d = 0;
       int ascii = 0;
       int packetcount = 0;
       int reconstructbuffer = 0;
       int TxScanFlag = 0;

       char NewArray[] = null;
       String outputData = new String();
       String FinalOutputData = new String();

       ArrayList Reconstruct = new ArrayList(); // will use array
       list Reconstruct to get back all information
       String FileName = new String();

       HandshakePkt = reader.readPacket(); // read the packet
       count for 1 time only

       // If a redundant handshake packet or an extra data
       packet, drop packet
       while((HandshakePkt[5] == buffer[5]) || (HandshakePkt[17]
       != 69))
       {
          HandshakePkt = reader.readPacket(); // read the packet
          count for 1 time only
          //System.out.print("\nExtra Data Packet or Not a
          Handshake Packet (dropped)\n");
       }
       char optionSelect = (char) HandshakePkt[7];

       sequenceNo = HandshakePkt[5];  // Update sequence number
```

```
//System.out.print("Received Sequence No: "
+Integer.toHexString(sequenceNo));

//System.out.print("\nTest " +HandshakePkt[6]);
//HandshakePkt[6] = Packet number
//System.out.print("\nSystem Terminal ID:" +(char)
TerminalID);

if(HandshakePkt[6] == TerminalID)
{
  System.out.print("\nRepeated Packet Dropped\n");
}


if(HandshakePkt[6] != (TerminalID)) // HandshakePkt[6] is
the Terminal ID
{
  //System.out.print(" This message is from other terminal
  ");

  System.out.println("\nOption Selected\t: "
  +optionSelect);

  int FileNameSize = HandshakePkt[8];      // This is to
  convert char in HandshakePkt into integer
  TxScanFlag = HandshakePkt[9];            // This
  is to signify whether heartbeat scan is requested or
  initiated

  // if(FileNameSize != 0)
  //   System.out.print("\nFileName Size\t: "
       +(char)FileNameSize);

  // ---------------------------------------------------
  // Find out size of array & number of packets needed to
     decode

  int arraysize = ((HandshakePkt[14] & 0xFF) <<16)
  + ((HandshakePkt[15] & 0xFF) << 8)
  + (HandshakePkt[16] & 0xFF);

  int k = ((HandshakePkt[10] & 0xFF) << 8)
  + (HandshakePkt[11] & 0xFF);

  int n = ((HandshakePkt[12] & 0xFF) << 8)
  + (HandshakePkt[13] & 0xFF);

  byte encoded = HandshakePkt[9];

  Integer.toHexString(HandshakePkt[i]) );


 // ------------------------------------------------------
```

```
if (encoded == 0)
{
  // -----------------------------------------------------
  // FEC Decoding Process for all incoming packets
  int packetsize = 26;                             // 29
  bytes minus 3 bytes overhead

  int block = k * packetsize;                      //
  Image block size
  int numblocks = (arraysize / block) + 1;         //
  Number of blocks we will Rx
  if ((arraysize % block) == 0)
     numblocks = numblocks - 1;
  int[][] receiverIndex = new int[numblocks][n];   //
  Index for all packets received

  // create our fec code
  FECCode fec =
  FECCodeFactory.getDefault().createFECCode(k,n);

  System.out.println("Reading in "+(n*numblocks)+"
  Encoded Packet(s)");
  // ------------------------------------------------------
  // Need to collect all incoming packets now and store
  in a byte array
  byte[][] received = new byte[numblocks][n*packetsize];
  // Storage for data read in
  int[][]     expected     =     new     int[numblocks][n];
  // Keeps track of which data read in
  byte               EOP1               =               0;
  //  Flag for terminating packet
  byte EOP2 = 0;
  byte EOP3 = 0;
  int redundant = 0;
  boolean condition = true;
  int packetcounter = 0;
  int[]     blockindex     =     new     int[numblocks];
  // Keep track of number of packets read ...

  // in for each block
  for              (int              i=0;i<numblocks;i++)
  // zero out  (necessary ?)
  blockindex[i] = 0;

  try
  {
    buffer             =             reader.readPacket();
    // Read in packet
    EOP1 = buffer[6];
    EOP2 = buffer[7];
    EOP3 = buffer[8];
    int          blocknum          =          buffer[5];
    // Find out block number
    int   packetnum   =   ((buffer[6]   &   0xFF)   <<   8)
    // Find out packet number
```

```
+ (buffer[7] & 0xFF);
while          (buffer[17]           ==           69)
// Catching extra handshake pckts
{
   buffer            =            reader.readPacket();
   // Read in packet
   EOP1 = buffer[6];
   EOP2 = buffer[7];
   EOP3 = buffer[8];
   blocknum                =                buffer[5];
   // Find out block number
   packetnum   =   ((buffer[6]   &   0xFF)   <<   8)
   // Find out packet number
   + (buffer[7] & 0xFF);
}

while ((packetnum > (n-1)) || (blocknum > (numblocks-
1)))
{
  buffer = reader.readPacket();                  //
  Read in packet
  EOP1 = buffer[6];
  EOP2 = buffer[7];
  EOP3 = buffer[8];
  packetnum = ((buffer[5] & 0xFF) << 8)          //
  Find out packet number
  + (buffer[6] & 0xFF);
}

condition = (EOP1==69) && (EOP2==69) && (EOP3==69);

while (condition != true)
{
  if (buffer.length < 34)          // Trying to catch
  incomplete packets
  {
    buffer = reader.readPacket();                  //
    Read in packet
    EOP1 = buffer[6];
    EOP2 = buffer[7];
    EOP3 = buffer[8];
    packetnum = ((buffer[5] & 0xFF) << 8)          //
    Find out packet number
    + (buffer[6] & 0xFF);
    condition = (EOP1==69) && (EOP2==69) &&
    (EOP3==69);

    while ((packetnum > (n-1)) || (blocknum >
    (numblocks-1)))
    {
      buffer = reader.readPacket();
      // Read in packet
      EOP1 = buffer[6];
      EOP2 = buffer[7];
      EOP3 = buffer[8];
```

```
        packetnum = ((buffer[5] & 0xFF) << 8)
        // Find out packet number
        + (buffer[6] & 0xFF);
        condition = (EOP1==69) && (EOP2==69) &&
        (EOP3==69);
    }
}


expected[blocknum][packetnum] = 1;
int countbyte = 0;
packetcounter++;

sequenceNo = (byte) packetnum;                    //
Update sequence No

for (int i = 8; i < 34 ; i++)                     //
Strip off headers
{
   received[blocknum][(packetsize…
   *blockindex[blocknum]) + (i-8)] = buffer[i];
}   // End for loop


receiverIndex[blocknum][blockindex[blocknum]] =
packetnum;
blockindex[blocknum]++;
buffer = reader.readPacket();
// Read in next packet
EOP1 = buffer[6];
// Check if terminating packet
EOP2 = buffer[7];
EOP3 = buffer[8];
condition = (EOP1==69) && (EOP3==69) && (EOP3==69);
blocknum = buffer[5];
// Find out block number
packetnum = ((buffer[6] & 0xFF) << 8)
// Find out packet number
+ (buffer[7] & 0xFF);
if (condition == true)
{
  blocknum = 0;
  packetnum = 0;
}

while ((packetnum > (n-1)) || (blocknum >
(numblocks-1)))
{
  buffer = reader.readPacket();
  // Read in packet
  EOP1 = buffer[6];
  EOP2 = buffer[7];
  EOP3 = buffer[8];
  packetnum = ((buffer[5] & 0xFF) << 8)
  // Find out packet number
```

```
         + (buffer[6] & 0xFF);
         condition = (EOP1==69) && (EOP2==69) &&
         (EOP3==69);
         if (condition == true)
         {
            blocknum = 0;
            packetnum = 0;
         }
      }
      System.out.println(blocknum + " " + packetnum);
      while ((expected[blocknum][packetnum] == 1) &&
      (condition != true))
      {

      //System.out.print("\nRedundant Data Packet
      dropped\n");
      redundant++;
      buffer = reader.readPacket();
      EOP1 = buffer[6];
      // Check if terminating packet
      EOP2 = buffer[7];
      EOP3 = buffer[8];
      blocknum = buffer[5];
      // Find out block number
      packetnum = ((buffer[6] & 0xFF) << 8)
      // Find out packet number
      + (buffer[7] & 0xFF);

      //System.out.println(blocknum+" "+packetnum);
     condition = (EOP1==69) && (EOP3==69) && (EOP3==69);
     if (condition == true)
     {
        blocknum = 0;
        packetnum = 0;
     }
   }  // end redundant packet check
 }  // End try
 } catch (IOException e) {}
 System.out.println(packetcounter +" out of " +
 (n*numblocks) + " packets received");
 System.out.println(redundant + " redundant packets");

 // ---------------------------------------------------------
 // Need to find out if we have enough packets read in for
 each block to decode

 boolean ok = true;
 for (int i=0; i<numblocks; i++)
 if (blockindex[i] < k)
    ok = false;


 // ---------------------------------------------------------

if (ok == true)  // Means enough data read in
```

```
{
   // Decoding Process
   System.out.println("Message Received: Decoding...");
   byte[]     received2     =     new     byte[arraysize];
   // All data minus padding
   Buffer[][]  receiverBuffer  =  new  Buffer[numblocks][n];
   // k subset of packets

   for (int z=0; z < numblocks; z++)
   {

      System.out.println("Decoding Block: " +(z+1)+" of "
      +numblocks);
      //create our Buffers for the encoded data
      for (int i = 0; i < n; i++)
      {
         receiverBuffer[z][i] = new Buffer(received[z],
         i*packetsize, packetsize);
      }

      //finally we can decode

      fec.decode(receiverBuffer[z], receiverIndex[z]);

    }

    // Now in the received array we have the decoded
    blocks of data but with padding
    // Need to remove the padding

    if (numblocks==1)

      System.arraycopy(received[0], 0, received2, 0,
      arraysize);
    else
    {
       for (int i=0; i< (numblocks -1); i++)
       {

          System.arraycopy(received[i], 0, received2 ,
          i*block, block);
       }
       System.out.println("");

       System.arraycopy(received[(numblocks -1)], 0,
       received2, ((numblocks-1)*block)
       , block - ((numblocks*block)- arraysize));
    }

    // End decoding process
    // -----------------------------------------------------

    if(optionSelect == '1')
    {
      // Print out the message
     System.out.println("TEXT MSG");
     System.out.println("Total Characters: " +arraysize);
                              String t = new
     String(received2 , "Cp1252");

     System.out.println("\nMessage Received: " +t);
     }//End of IF loop for Option 1

     else if(optionSelect == '2')
     {
       System.out.println("Text File TX");

       // Let user choose save location
       JFileChooser fileChooser = new JFileChooser(".");
       int status = fileChooser.showSaveDialog(null);
       String filename = new String();
       filename = "C:/Documents and
       Settings/Administrator/My Documents/My
       Pictures/test.jpg";
       if (status == JFileChooser.APPROVE_OPTION)
       {
          File selectedFile =
          fileChooser.getSelectedFile();

          System.out.println("Saving to: " +
          selectedFile.getPath());
          filename = selectedFile.getPath();
       }

       File fileOut;
       BufferedInputStream in = null;
       BufferedOutputStream out = null;

       try
       {
         out = new BufferedOutputStream(new
         FileOutputStream(filename));
         out.write(received2);
       }
       catch (IOException ex)
       {
         ex.printStackTrace();
       }
       finally
       {            // closing the stream
         try
         {
          if (in != null) in.close();
            if (out != null) out.close();
         }
         catch (IOException ex) { ex.printStackTrace(); }
      }

      }  // End option 2 if portion
```

```
                   else if(optionSelect == '3')
                   {
                     System.out.println("Image TX");
                     // Let user choose save location
                     String filename = new String();
                    filename = "C:/Documents and
                    Settings/Administrator/My Documents/My
                    Pictures/test.jpg";
//                  JFileChooser fileChooser = new JFileChooser(".");
//                  int status = fileChooser.showSaveDialog(null);
//                  if (status == JFileChooser.APPROVE_OPTION)
//                  {
//                    File selectedFile =
                      fileChooser.getSelectedFile();
//
                      System.out.println("Saving to: " +
                      selectedFile.getPath());
//                    filename = selectedFile.getPath();
//                  }

                    File fileOut;
                    BufferedInputStream in = null;
                    BufferedOutputStream out = null;

                    try   // Write Image to a file
                    {
                       InputStream input = new
                       ByteArrayInputStream(received2);
                       BufferedImage bi =
                       javax.imageio.ImageIO.read(input);
                       File outputfile = new File(filename);
                       ImageIO.write(bi, "jpeg", outputfile);
                    }
                    catch (IOException ex)
                    {
                       ex.printStackTrace();
                    }
                    finally
                    {            // closing the stream
                      try
                      {
                        if (in != null) in.close();
                          if (out != null) out.close();
                      }
                      catch (IOException ex) { ex.printStackTrace(); }
                    }

                 }  // End option 3 if portion
              }  // end if true statement
              else System.out.println("Not enough packets read in");

           } // End if encoded
           //  -----------------------------------------------------
          else if (encoded == 1)
          {
```

```
 //   ---------------------------------------------------------
 //  ---------------------------------------------------------
 //  Now for non-encoded data

 // ----------------------------------------------------------
int count = k;
int packetcounter = 0;
int packetnum = 0;
byte EOP1 = 0;                          //  Flag
byte EOP2 = 0;
byte EOP3 = 0;
boolean condition = true;
System.out.println("Reading in "+k+" Packet(s)...");
byte[] received = new byte[k*27];
byte[] received2 = new byte[arraysize];
int[] expected = new int[k];
int countbyte = 0;
int redundant = 0;

try
{
   buffer = reader.readPacket();                      // Read
   in packet
   if (buffer[17] == 69)                               // Catch
     for extra handshake packet
   buffer = reader.readPacket();
   EOP1 = buffer[6];
   EOP2 = buffer[7];
   EOP3 = buffer[8];
   packetnum = ((buffer[5] & 0xFF) << 8)              // Find
   out packet number
   + (buffer[6] & 0xFF);
   while (buffer[17] == 69)                           //
   Catching extra handshake pckts
   {
     buffer = reader.readPacket();                   // Read in
     packet
     EOP1 =  buffer[6];
     EOP2 = buffer[7];
     EOP3 = buffer[8];
     packetnum = ((buffer[5] & 0xFF) << 8)          // Find
     out packet number
     + (buffer[6] & 0xFF);
   }

   condition = (EOP1==69) && (EOP2==69) && (EOP3==69);

   while (condition != true)
   {
     if (condition != true)
     {
      if (buffer.length < 34)          // Trying to catch
      incomplete packets
                                      {
```

```
  buffer = reader.readPacket();                  // Read
  in packet
  EOP1 = buffer[6];
  EOP2 = buffer[7];
  EOP3 = buffer[8];
  packetnum = ((buffer[5] & 0xFF) << 8)          // Find
  out packet number
  + (buffer[6] & 0xFF);
}

expected[packetnum] = 1;
packetcounter++;
sequenceNo = (byte) buffer[5];                // Update
sequence No

for (int i = 7; i < 34 ; i++)                 // Strip off
headers
{

   //System.out.println(packetnum + " " +countbyte + " " +
   i + " " + redundant);
   received[countbyte] = buffer[i];
   countbyte++;

}   // End for loop

buffer = reader.readPacket();               // Read in next
packet
EOP1 = buffer[6];                          // Check if
terminating packet
EOP2 = buffer[7];
EOP3 = buffer[8];
packetnum = ((buffer[5] & 0xFF) << 8)      // Find out
packet number
+ (buffer[6] & 0xFF);
condition = (EOP1==69) && (EOP3==69) && (EOP3==69);
if (condition == true)
packetnum = 0;
} // end if

while ((expected[packetnum] == 1) && (condition != true))
// Catching redundant packets
{

  //System.out.print("\nRedundant Data Packet dropped\n");
  redundant++;
  buffer = reader.readPacket();
  EOP1 = buffer[6];                         // Check if
  terminating packet
  EOP2 = buffer[7];
  EOP3 = buffer[8];
  packetnum = ((buffer[5] & 0xFF) << 8)              //
  Find out packet number
  + (buffer[6] & 0xFF);
  condition = (EOP1==69) && (EOP3==69) && (EOP3==69);
```

```
       if (condition == true)
          packetnum = 0;
      }
   }  // End while
   System.arraycopy(received, 0, received2, 0, arraysize);
} catch (IOException e) {}

System.out.println(packetcounter+ " out of " +k+ " packets 
received");
System.out.println(redundant+  " redundant packets");


// -----------------------------------------------------------
// Non-encoded packets received, now to save/display

if (packetcounter == k)
{
  if(optionSelect == '1')
  {
    // Print out the message
    System.out.println("TEXT MSG");

    System.out.println("Total Characters: " +arraysize);
    String t = new String(received2 , "Cp1252");

    System.out.println("\nMessage Received: " +t);
    }//End of IF loop for Option 1

    else if(optionSelect == '2')
    {

    System.out.println("Text File TX");

    // Let user choose save location
   JFileChooser fileChooser = new JFileChooser(".");
   int status = fileChooser.showSaveDialog(null);
   String filename = new String();
   filename = "C:/Documents and Settings/Administrator/My 
   Documents/My Pictures/test.jpg";
   if (status == JFileChooser.APPROVE_OPTION)
   {
      File selectedFile = fileChooser.getSelectedFile();
      System.out.println("Saving to: " + 
      selectedFile.getPath());
      filename = selectedFile.getPath();
   }

   File fileOut;
   BufferedInputStream in = null;
   BufferedOutputStream out = null;

   try
   {
      out = new BufferedOutputStream(new 
      FileOutputStream(filename));
```

```
        out.write(received2);
     }
     catch (IOException ex)
     {
       ex.printStackTrace();
     }
   finally
   {            // closing the stream
     try
     {
       if (in != null) in.close();
       if (out != null) out.close();
     }
     catch (IOException ex) { ex.printStackTrace(); }
   }
}  // End option 2 if portion

else if(optionSelect == '3')
{
     System.out.println("Image TX");
     // Let user choose save location
     String filename = new String();
     filename = "C:/Documents and Settings/Administrator/My
     Documents/My Pictures/test.jpg";
     // JFileChooser fileChooser = new JFileChooser(".");
     // int status = fileChooser.showSaveDialog(null);
     // if (status == JFileChooser.APPROVE_OPTION)
     // {
     //   File selectedFile = fileChooser.getSelectedFile();
     //
         System.out.println("Saving to: " +
         selectedFile.getPath());
     //  filename = selectedFile.getPath();
     // }

    File fileOut;
    BufferedInputStream in = null;
    BufferedOutputStream out = null;

    try   // Write Image to a file
    {
       InputStream input = new ByteArrayInputStream(received2);
       BufferedImage bi = javax.imageio.ImageIO.read(input);
       File outputfile = new File(filename);
       ImageIO.write(bi, "jpeg", outputfile);
    }
    catch (IOException ex)
    {
       ex.printStackTrace();
    }
    finally
    {            // closing the stream
      try
      {
         if (in != null) in.close();
```

```
                if (out != null) out.close();
             }
             catch (IOException ex) { ex.printStackTrace(); }
           }

         }  // End option 3
      }  // end if not enough packets
      else if (packetcounter < k)
      System.out.println("Not enough packets received to process");
     }   // End if non-encoded loop
    } //End of Terminal ID Check Loop

   } // End of while infinite loop

  } // End of try statement
  catch (IOException e)
  {
     System.err.println("Error on " + reader.getName() + ": " + e);
  }
 }  // End main Class

}  // End SnailListenTest_fec
```

LIST OF REFERENCES

[1] Chee-Yee Chong, “Sensor Networks: Evolution, Opportunities, and Challenges," http://www-net.cs.umass.edu/cs791_sensornets/papers/chong.pdf, Last accessed 01 Nov 2008.

[2] "Implementing VOIP over Wireless Network," Retrieved from the Alvarion site: http://www.alvarion.com/upload/contents/291/VoIP over wireless networks 060706.pdf. Last accessed 01 Nov 2008.

[3] University of California, Berkeley, “Forward Error Correction in Sensor Networks,” Jaein Jong and Cheng Tien Ee, http://webs.cs.berkeley.edu/papers/FEC_report.pdf, Last accessed 01 Nov 2008.

[4] DongGuk University, Seoul, Korea, “An Adaptive FEC Code Control Algorithm for Mobile Wireless Sensor Networks,” Jong-Suk Ahn, Seung-Wook Hong, and John Heidemann, http://www.isi.edu/~johnh/PAPERS/Ahn05a.pdf, Last accessed 01 Nov 2008.

[5] Terry D. Norbraten, “Utilization of Forward Error Correction (FEC) Techniques with Extensible Markup Language (XML) Schema-Based Binary Compression (XSBC) Technology,” Master’s Thesis, Naval Postgraduate School, Monterey, California, Dec 2004.

[6] Injong Rhee, Ajit Warrier, Mahesh Aia, and Jeongki Min, “Z-MAC: A Hybrid MAC for Wireless Sensor Networks,” IEEE Communications Magazine, Volume 16, Issue 3, pp. 511–524, June 2008.

[7] T. Van Dam and K. Langendoen. An Adaptive Energy Efficient MAC Protocol for Wireless Sensor Networks. In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, CA, Nov 2003. (Tmac).

[8] W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated adaptive sleeping for wireless sensor networks. IEEE/ACM Trans. Netw., 12(3):493{506, 2004. (Smac).

[9] Yow Thiam Poh, “Tunneled Data Transmission Over Wireless Sensor Networks,” Master’s Thesis, Naval Postgraduate School, Monterey, California, Dec 2007.

[10] John A. Stankovic, “Wireless Sensor Networks,” IEEE Communications Magazine, Volume 41, Issue 10, pp. 92– 95, Oct 2008.

[11] Hong Kong Baptist University, “Resilient Proactive Data Transmission in Wireless Sensor Networks,” Yingqi Xu, Jianliang Xu, and Wang-Chien Lee, http://www.comp.hkbu.edu.hk/~xujl/Papers/infocom2007.pdf, Last accessed 01 Nov 2008.

[12] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Transactions on Information Theory, IT(13):260–269, 1967.

[13] L. Rizzo. Effective erasure codes for reliable computer communication protocols. SIGCOMM Comput. Commun. Rev., 27(2):24–36, 1997.

[14] Crossbow Technology Inc, “Smart Dust Training Seminar,” San Jose, Apr 19-20, 2005.

[15] Georgia Institute of Technology, “A Survey on Sensor Networks,” Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci, http://www-net.cs.umass.edu/cs791_sensornets/papers/akyildiz2.pdf. Last accessed 01 Nov 2008.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Department of Electrical Engineering
   Naval Postgraduate School
   Monterey, California

4. Professor John C. McEachen, Code EC/Mj
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

5. Professor Murali Tummala, Code EC/Tu
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California