

Analysis of Wireless Data Transmission Characteristics

Rachel Rubin

Faculty Mentor: David Culler

Graduate Mentor: Robert Szewczyk

August 4, 2000

Abstract

The cooperation between the Ninja and SmartDust projects at the University of California at Berkeley has produced an operating system that runs on a set of networked sensors. The sensors are connected through a wireless radio. The error characteristics in the data transmissions are not well known or well understood; the knowledge of the error patterns will allow the system designers to pick an encoding scheme for the packets. We present a study of these error characteristics. In the process of characterizing these errors, we discovered several problems within Tiny OS. The error characteristics show that the current encoding scheme used by the designers of TinyOS is inappropriate.

1 Introduction

1.1 The Ninja Project

The Ninja project works toward the design and implementation of robust, scalable, distributed Internet services[1]. Its architecture, shown in Figure 1, is composed of four basic elements: bases, units, active proxies, and paths. *Bases* are scalable cluster environments which were designed to run the core of services. The *units* are the end clients within the system; these are numerous devices of varying capabilities trying to exchange information with the Internet. *Active proxies* are the transformational elements that are used for device- or service-specific adaptation. Finally *paths* form an abstraction used to compose the elements of the infrastructure to deliver exchange the information between a base and a unit.

The Ninja project seeks to design a service, an embedded software that exports a network-accessible programatic interface and allows for strong operational guarantees. This service is run on a base and must support the units. These units may have limited accessibility, connectivity and computational ability.

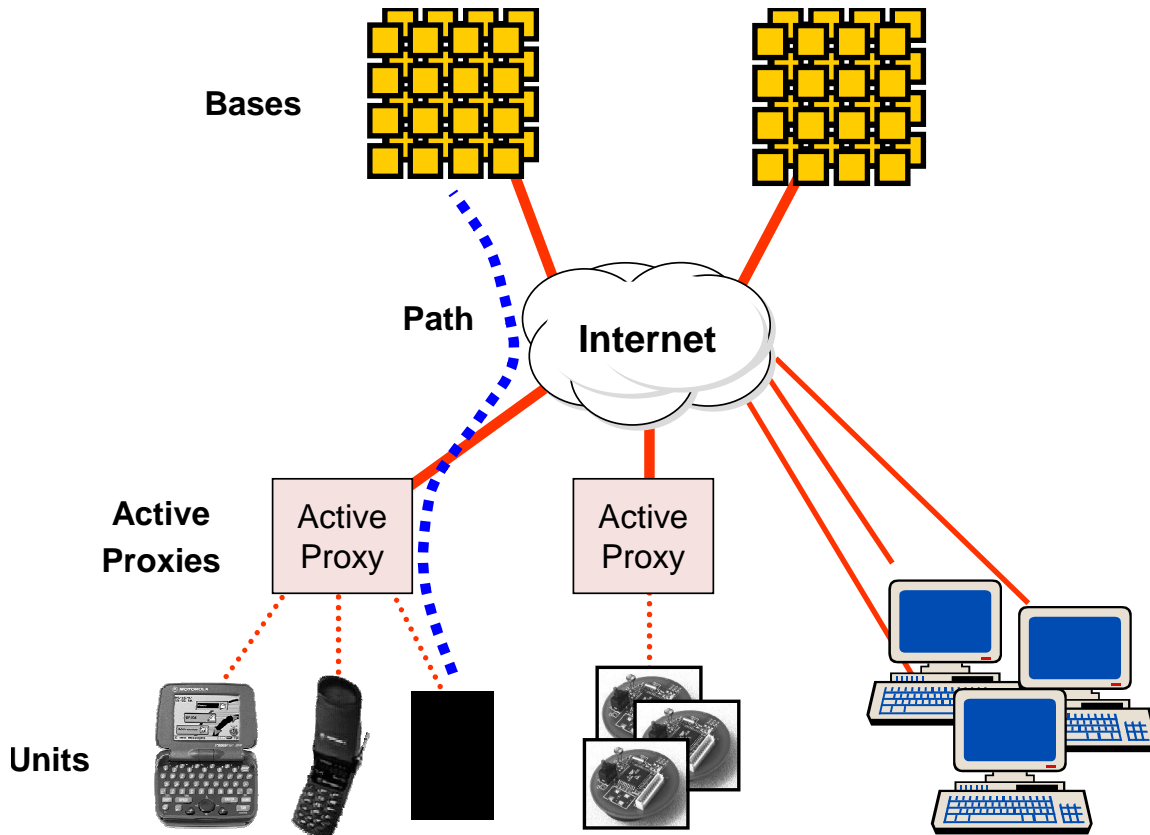


Figure 1: Ninja architecture

The service gains robustness through a process called conditioning the service. This is done by selecting an appropriate programming model and concurrency strategy to maintain the resources. The service is viewed as a series of stages separated by variable-latency operations, such as receiving a requested file from the file system or returning a response to a browser, by the programming model. These stages share data using pass-by-value semantics and this constraint allows stages to be separated from each other, perhaps across physical machine boundaries. The task-driven composition of the stages works well with the event-driven devices the Ninja project is working with and allows the architecture to scale to the point where it can handle thousands of concurrent tasks.

1.2 Networked Sensors

The Internet services of the future will have to support not only a wide variety of clients browsing web pages, but also a much tighter integration with the physical world. This will be accomplished by integrating real-world measurements from a wide variety of sensors. The SmartDust project [3] at UC Berkeley is creating many small scale networked sensors. Their goal is to integrate the sensing, with processing, communication and power modules in a 1 mm³; the current prototype built with commercial parts is about the size of a silver



Figure 2: Hardware picture

dollar (see Figure 2). A part of the Ninja project is to develop a software architecture for programming these devices.

In order to meet the goals of size and cost, the networked sensors will have limited computational resources, low storage capability, no human interface [2]. A current prototype of a networked sensor developed by the SmartDust group contains a radio, a microcontroller and photo and light sensors. The sensor, also called a mote¹, can form ad hoc wireless networks, take measurements, and report information to services for analysis. This device is used as the unit in the Ninja architecture in all of the work described below.

1.3 Tiny OS

There are several requirements that a networked sensor must meet[2]. Limited amount of storage means that the software running on the sensor may not be able to do a lot of buffering. Thus rather than operating in request-response mode, the software system should be able to handle information flows. The constraints of cost and energy usage will limit the dedicated hardware resources available on the device; the software system must thus cope with concurrency intensive operations in spite of limited physical resources. The applications of networked sensors are expected to be quite diverse; the operating system for these devices must be able to support these multiple applications. Finally, the system as a whole must be robust to failure of its parts.

In order to meet these requirements, researchers at UC Berkeley developed a small micro-threaded operating system called Tiny-OS. It is an event-driven system composed of plug-gable software modules. It is designed to handle a high degree of concurrency despite limited hardware resources. The implementation language for the system is C.

The basic abstraction in Tiny OS is a software *component*. The component consists of four interrelated parts: *command handlers*, *event handlers*, a fixed-size *frame* and a bundle

¹The current generation of these devices is clearly too large to earn the name “dust”. Currently a more fitting name could be a smart pebble, the Smart Dust felt it was appropriate to call this device a “dust mote”.

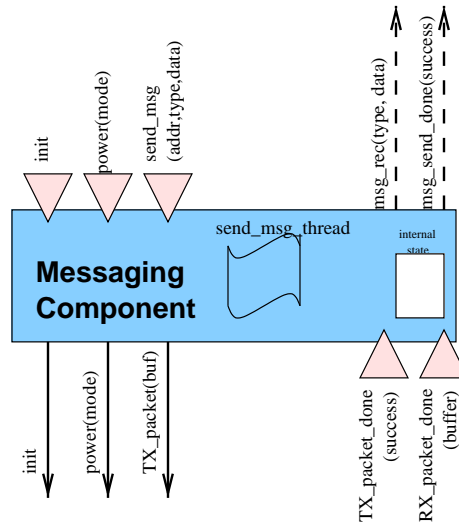


Figure 3: Sample Component. Each component declares strong interfaces: it explicitly defines not only the commands and events that it handles, but also the commands and events it uses. This feature makes it quite easy to combine individual components into a complete application using graphical tools.

of simple *tasks*. Figure 3 shows the graphical representation of a component. Commands are provide the main interface to each component, and should in general be nonblocking. The events provide a mechanism for completion notification. The statically allocated frame keeps the state of each component; the static allocation allows for verification of the memory requirements at compile time and avoid the overhead of memory management at runtime. The tasks embody the main computation of each component. By design, tasks run to completion with respect to other tasks, but can be preempted by events. The complete system configuration consists of a graph on components and a lightweight task scheduler.

Components fall into three categories: hardware abstractions, synthetic hardware and high level software [2]. Hardware abstraction maps physical hardware onto the component model. Synthetic hardware simulates the behavior of real hardware. The software components perform control, routing and data transformations. This component structure allows for the breakdown of the hardware/software boundary by allowing easy migration across it. The event based model is complementary to the underlying hardware.

The interaction of threads and events within Tiny OS is known as two-level scheduling. As such it meshes well with a communication model known as Active Messages. The main idea behind this communication model is to match the communication primitives to the underlying hardware, and integrate communications and processing. It was developed in the context of high performance parallel computing to minimize latency; however the high efficiency of the model and low buffering requirements make it a good candidate for communication between networked sensors. Each active message contains the name of a user-level handler that is invoked on a target node when it arrives and arguments are passed in in a data payload.

2 Project

2.1 Purpose

The purpose of the project this summer was to characterize the errors in the radio transmission of the remote sensors. Although the radio is used often, its error characteristics are not well known or understood. This would allow an encoding scheme to be picked to match the error characteristics. As a byproduct of this project, however, the analysis of the types and placements of errors allowed conclusions to be drawn about the locations of the bugs in Tiny OS.

2.2 Hardware

The notes that the Ninja project works with consist of a microcontroller with internal flash program memory, data SRAM and data EEPROM. These are connected to actuator and sensor devices including LEDs, a low-power radio transceiver, an analog photo-sensor, a digital temperature sensor, a serial port and a small coprocessor unit [2].

The radio is the most important part of the sensor for this project, since the data examined is all transmitted via radio. It is an asynchronous input/output device with real-time constraints [2]. The radio has no buffering, so each bit that passes through the radio must be dealt with as it is sensed. The radio is an RF Monolithics TR1000 operating in 916.50 MHz band[4]. The radio is designed for low-power, short-range wireless data communications. Its design has several implications on the Tiny OS: in particular, the radio exposes raw bit-level interface, and the software needs to ensure the synchronization between sender and receiver, as well as reading every bit within tight real-time constraints.

2.3 Packets

The sensors sent information which was interpreted in 60-byte packets. Packets were set up to conform to the standard set by Active Messaging. On the sender side, Tiny OS exposes asynchronous interface to send Active Messages. On the receiver side, the system accepts a message incoming from the network, and dispatches it to the handler for which it was addressed.

The messages have a standard format: the packets contains a fixed header, and a data payload. Currently, the payload is 20 bytes long and is replicated 3 times to ensure robustness against errors. A voting scheme is used on a byte-by-byte basis to recover the data. Within the payload there is an address, and a dispatch identifier. The dispatch routine invoked by the handler is generated at compile time based on the message handlers present, which eliminates the need for handler registration mechanisms.

2.4 Code

Code was developed that would read in bits from the mote and compare them with the bits that were expected given that packet type. The bits were read in in sixty character chunks or packets. These packets were compared with the packet that was expected given the opening sequence of bits.

The bits, once they were read in, were dealt with in two different ways: encoded and decoded. The encoded bits were the ones read in from the mote-base. However, the program also needed to deal with the data in a decoded fashion, where the expected behavior was known. The decoding was done by taking two consecutive eight-bit characters and making them into one character. The stream that was being dealt with was reduced from 480 bits to 240 bits and the behavior of this stream was known. This allowed the packets to be classified and dealt with appropriately.

There were two different packets that the sensors could send. One, beginning with an `0xff`, described a routing path and did not change. The other transmitted packet began with a `0x05` and `0x06`. However, as opposed to the other packet-type, the contents of this packet changed in a known pattern. The hex-values of the tenth and twentieth decoded character incremented by 11. When the hex-value of this decoding reached `0xff`, the value would wrap around and the ninth and tenth decoded character's hex value would increase by one. A similar event occurred in the thirtieth decoded bit, except that the hex-value increased by 16. A known but changing bit pattern was generated, which made it easier to detect bit errors. Since the bits were constantly changing, errors would show up and not be hidden in a constant stream; the pattern also made it easier to see if any packets were missed.

3 Experiment

3.1 Setup

A mote-base was attached to a PC via the serial port and was used to gather data. A remote sensor transmitted packets to the mote-base which were subsequently analyzed by the program. The remote sensor was placed at four distances: 2 inches, 10 inches, 21 inches and 58 inches. At these distances, the sensor was turned on and the program run until it read approximately 3000 packets. Only the dynamic packets were sent to the PC for analysis. However, since changes occurred, the behavior of the mote was variable and errors may occur.

3.2 Design

The error analysis tool, as opposed to be run on the motes, was a higher-level interface, or service, that was run on the bases. It was written in C. The code was structured simply, as shown by the flow chart in Figure 4 . After it checked its alignment, so that the beginnings

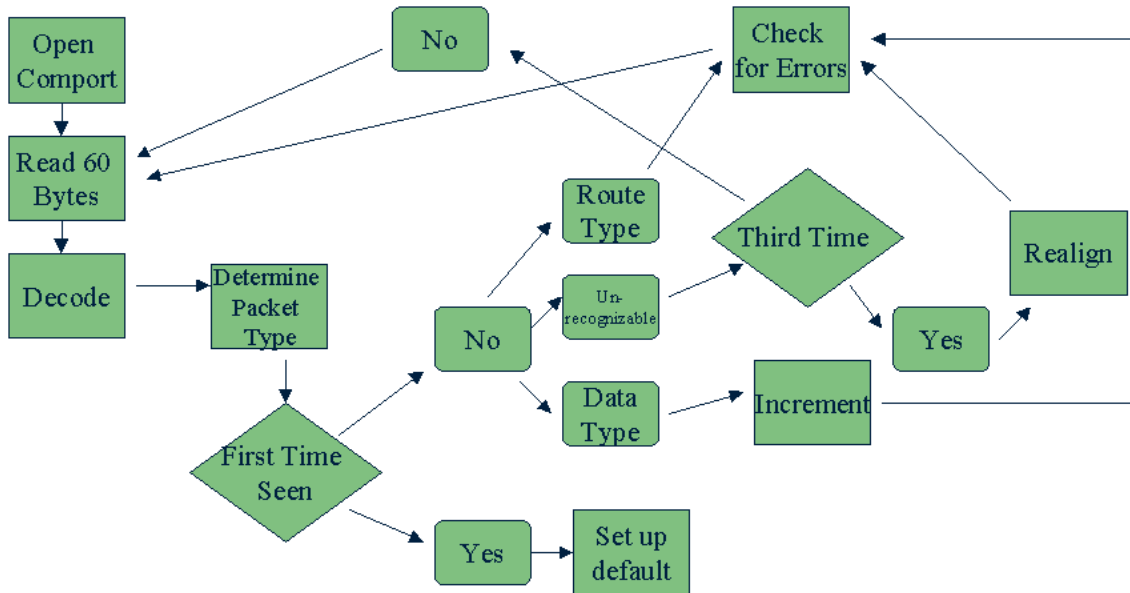


Figure 4: Flow chart of the code structure

of the packets were being read in at the correct time, the program would read in a sixty character chunk. The first decoded characters of this packet were examined and the packet type was determined. If the packet began with the 0x5 0x6 the correct, stored packet was incremented so that the packets were, in theory, identical. Then, the expected and received packets were compared bit by bit to determine if there was an error and, if so, where the error was. This process was repeated until the user hard stopped it.

3.3 Measurements

Several pieces of data were collected during this process. The data collected was the total number of bits seen, the total number of packets seen, the total number of bit errors, which bits were wrong and which way they were off, the number of errors turning up in each type of packet, the number of times each packet was seen, total number of packets with errors in them, the number of 0x05 0x06, or data, packets that came out of sequence, the position of each bit error, and the number of unrecognizable packets. After each packet was analyzed, the data and analysis was displayed so that the user could follow what was happening. The measurements were made across two independent variables - distance between the mote-base and the sensor and the time that the sensor had been running. Although there were two types of packets, only one was used for the experiments for simplicity sake.

3.4 Problems

One problem that was run across during the course of the experiments was that the measurements were taken with a faulty radio. This problem led to false initial conclusions being

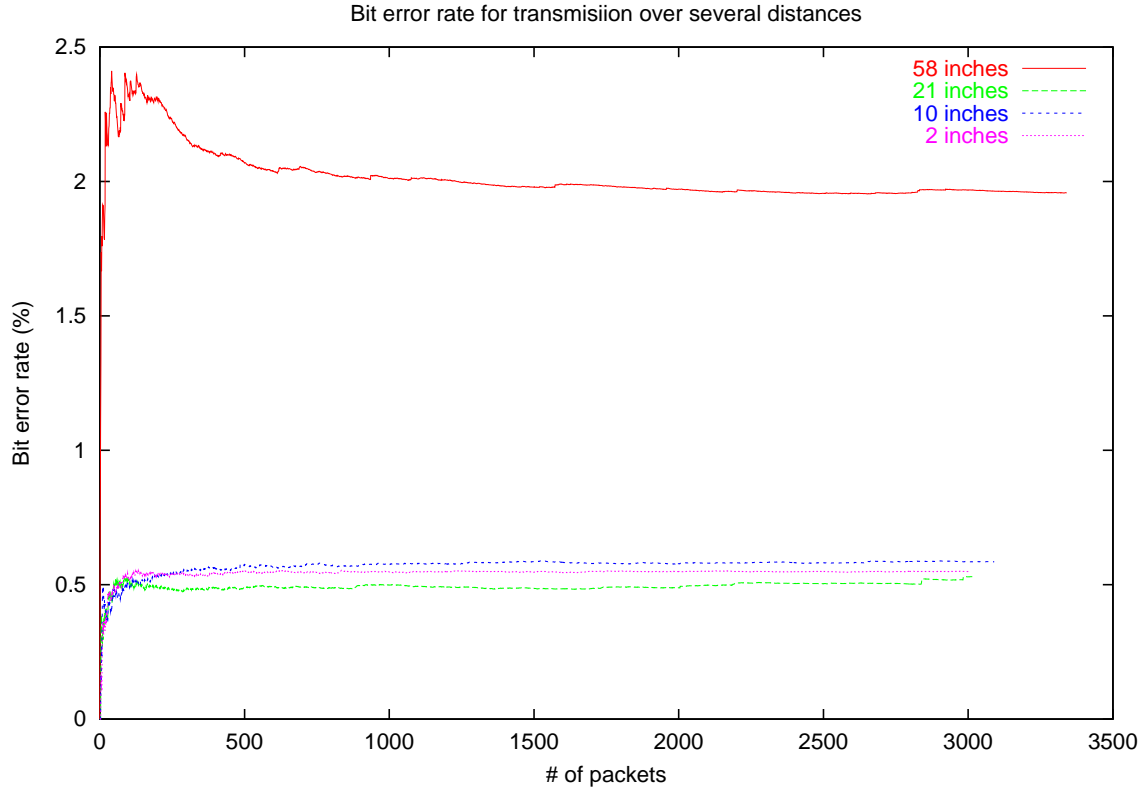


Figure 5: Bit error rate for transmission over several distances

drawn; on the other hand it also gave some insight into what would happen if a radio went bad. This is discussed in more detail below.

4 Results

4.1 Bit Errors

One of the major measurements that was taken was how many raw bit errors were present in the radio transmission. This data would give a rough idea on the accuracy of the radio transmissions.

Fewer errors were detected when the sensor was closer to the mote-base than when it is further away. However, this can be expected from the specs noted in the design of the radio because of the strength of the radio. However, an optimal distance from the sensor to the mote-base can be determined. Up to a point, the percent of bits in error hovers around .5 consistently throughout the run. When the radio is getting out of range, the bit error rate jumps to around two percent.

The bit error rate is consistent across time. For example, if the error rate levels out early on at .5%, that remains consistent throughout the run. While the error rate did vary from run

to run, this variation was fairly small within the normal range of the radio.

Originally, there was an increase in the bit error rate over time. This led to the conclusion that there was a problem in Tiny-OS, since the bit error rate should be constant across time. The bug was eventually tracked down. There was a race condition within the queue of Tiny OS which was causing bits to be changed. Using the information gathered, the bug was tracked down and fixed.

4.2 Bit Error Position and Characteristics

The bit errors are evenly distributed between runs. It is a random, even distribution from run to run. However, the errors in one run tend to cluster in certain bits, but the positions of these clusters vary from run to run. Interestingly enough, the most of the bit errors observed occur in a fixed bit within a payload. As we pointed out above, the current encoding scheme transmits a 20 byte payload 3 times; many of the bit errors are separated by 160 bits. For example, in one run there were nine errors at bit position twenty-six, one hundred and eighty-six and three hundred and forty-six. That finding suggests that there is some erroneous behavior within the Tiny OS which will have to be investigated further.

There is also a pattern to which way the bit changes. If a zero bit was expected but a one was received, it indicated that there was most likely interference in the transmission, most likely in the radio. If a zero bit were seen when a one was expected, it meant that there was an error causing the true-bit negate. Another explanation could be that there is a problem with the strength of the radio so that the high-frequency is not being received. As shown in Figure 6, over 65% of the errors occur when the bit received is a zero when it is supposed to be a one.

4.3 Percent Packets in Error

The percentage of packets that had bit errors in them remained constant throughout time. For the packets that were within the range of the radio, around fifty percent of the packets had some type of error. However, as the radio approached the outskirts of its range, almost all of the packets arrived with at least one error. This finding is consistent with the bit error rates and position characteristics described earlier.

4.4 Dropped Packets

There seems to be a problem with dropped packets. The sensor is not adding 11 consistently in the last of the decoded characters. The sequence at times gets completely off, it is at points going backwards and not always by 11. The rate that this error occurs is consistent, as the number grows linearly across time. However, in this case distance is a factor in the number of dropped packets. The further the sensor is to the mote-base, the more packets are dropped.

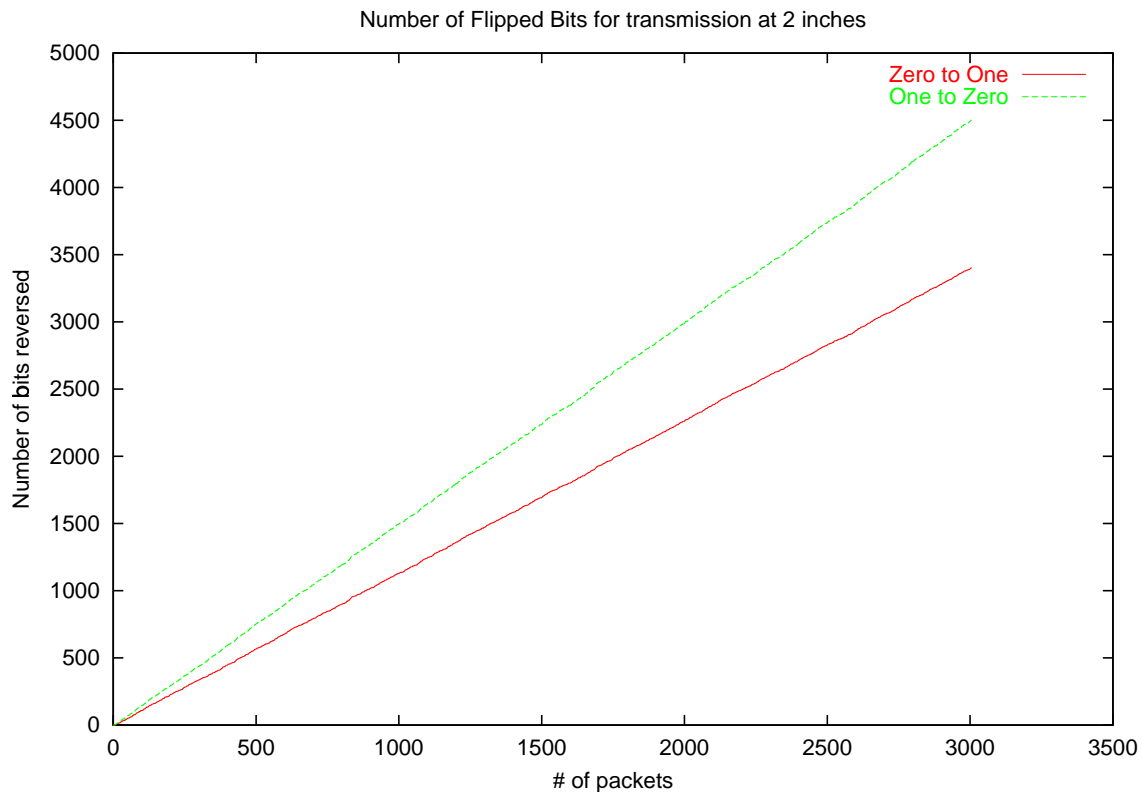


Figure 6: Types of bit errors.

4.5 Unrecognizable Packets

At times, packets would be seen that the sensor could not interpret. The frequency of the arrival of these packets increased as time progressed. Also, the frequency of these packets was greater when the distance was greater.

One of the major discoveries about Tiny-OS this summer is that the longer the mote runs, the more likely it becomes that an unrecognizable packet will be received. In earlier versions of the error code, this was an unrecoverable error. However, upon closer examination, it was noticed that the unrecognizable packets existed because at some point bytes were dropped and the short packet caused a misalignment of the packets. Because entire bytes were dropped, as opposed to just bits, it suggests that the problem is not in the transmission, but in Tiny-OS. When a patch was added to the code to realign the bytes, this became a negligible error. The patch will eventually be incorporated into the Tiny-OS system.

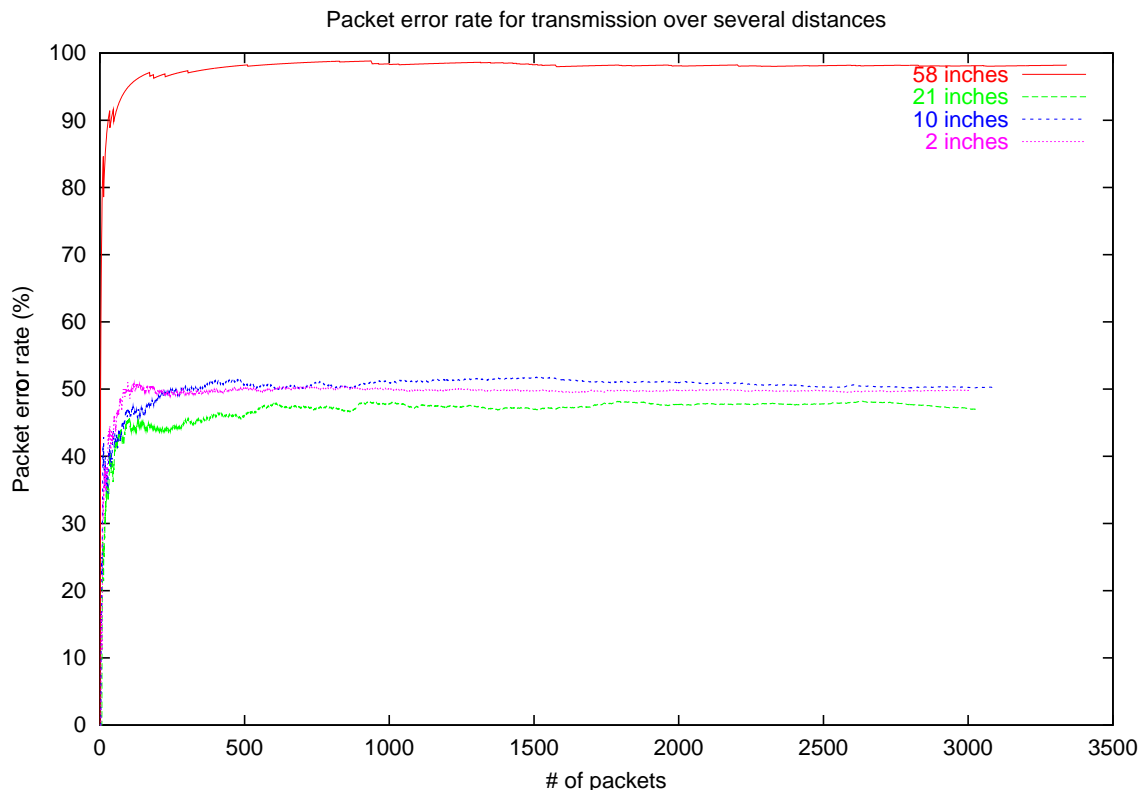


Figure 7: Percent of Packets Arriving With Errors

5 Conclusion

5.1 Bit Error Rate

One of the major implications of the finding this summer is that the bit error rate is not high enough to warrant the precautions that are being made to guard against it. The packets have a predictable distribution of bits in error. The errors in one run tend to cluster in certain bits, but the positions of these clusters vary from run to run. They seem to occur in the same position in the one hundred and sixty bit subdivision of each packet. For example, in one run there were nine errors at bit position twenty-six, one hundred and eighty-six and three hundred and forty-six. In all cases, there were not the same number of bit errors in complementary bit positions, but they were always close. This indicates that the expensive precaution of transmitting the data three times is not the correct algorithm to be using to ensure correctness.

There is also a pattern to which way the bit changes. If a zero bit was expected but a one was received, it indicated that there was most likely interference in the transmission, most likely in the radio. If a zero bit were seen when a one was expected, it meant that there was an error causing the true-bit negate. Over half the time the bit received is a zero when it is supposed to be a one which indicates a weakness in the radio transmission or, possibly, the mote itself.

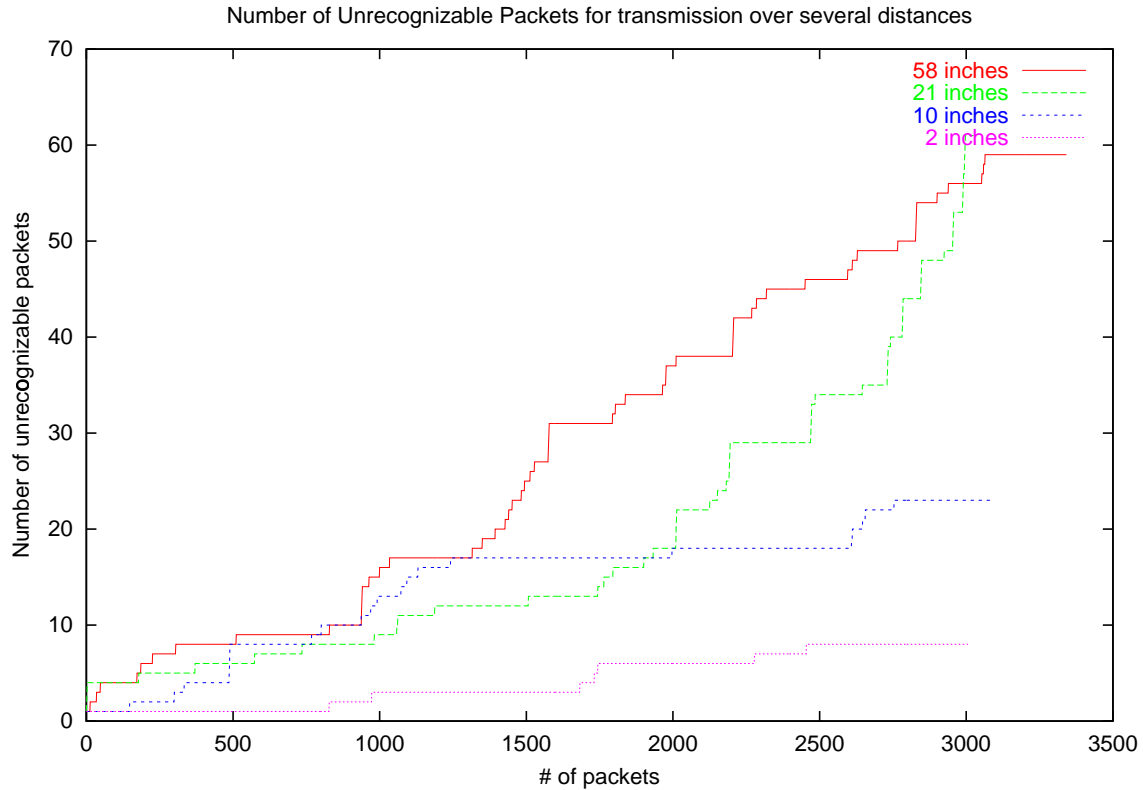


Figure 8: Number of Unrecognizable Packets

5.2 Tiny-OS

The project started because the TinyOS programmers observed many problems with radio transmission. The analysis of error pattern revealed that most of the perceived errors were caused by the bugs within the OS itself rather than problems with the radio transmission. Problems that were discovered using this tool included the eventual misalignment of the packets and the presence of race conditions.

5.3 Lessons Learned

The project began as an attempt to learn about errors in radio transmissions. However, although some minor characteristics were learned, the major implications of the project was discovering bugs in the operating system. As opposed to debugging a normal program, an operating system can not be perfected by looking through code. Typically, it involves analyzing some aspect of the operating system's behavior. Then, a hypothesis is developed that matches the observed behavior and changes are made to the system. The tool that was developed, which was supposed to discover characteristics of the radio actually allowed these hypotheses to be developed and the bugs to be removed.

6 Acknowledgements

I would like to thank Professor David Culler for giving me the opportunity to work in his lab. Of course, without the help and knowledge of my graduate mentor Robert Szewczyk and the other members of the lab, Jason Hill and Matt Welsh, I would not have been able to have a successful summer. I would also like to thank the organizers of the SUPERB program, especially Marie Mayne, Dr. Shelia Humphries, Monica Lin and Suzanne Kauer for putting together the program. Finally, I would like to thank my fellow SUPERB-ites for their friendship and support throughout the summer.

References

- [1] Steven D. Gribble et. al. The ninja architecture for robust internet-scale systems and services. *Special Issue of Computer Networks on Pervasive Computing*, 2000.
- [2] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, 2000.
- [3] K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999.
- [4] RF Monolithics. *TR1000 916 MHz Hybrid Transceiver*.