EC, GH
Project Number:EXC-0901

ALMOND

Acoustic Localization for Mobile Open-Source Network Deployment

A Major Qualifying Project Report
submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
by:

Scott Almquist

Muhammad Saleem

Daniel Skehan

Date: October 15, 2009

Approved: Professor Edward Clancy

Professor George Heineman

Approved for public release - distribution is unlimited

This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Abstract

Acoustic localization is a system of determining the position of a sound source via an array of acoustic sensors. Historically, acoustic localization has been performed by hardware that was custom designed for the task. But if it were possible to use commercially available cell phones as acoustic sensors, then there could be a substantial savings in non-recurring engineering costs and a significant speed up in the time to realization of a system.

This project used two Openmoko Freerunner cell phones to investigate the potential of a cell phone to perform acoustic localization. The phones were connected to a wireless network where they streamed audio data to a laptop computer to be processed.

The phone's ability to detect sounds exceeded the researchers' expectations. The time window at which a sound started was determined to within 68 microseconds, for a particular sound. Additionally, it was found that the phones were able to hear frequencies ranging from 100 hertz to 20 kilohertz.

One of the major implementation problems of an acoustic localization system is the synchronization of time between the sensors. A one millisecond timing error will result in a *minimum* of 0.17 meters of positional error. This project tested two methods for synchronizing time between the phones.

The first method of synchronization involved using a calibration sound that was equidistant from the phones. The time could then be changed based on when the sound was heard by each phone. Using this method, the phones were synchronized to within eight milliseconds for over five minutes in time.

The second method of synchronization involved using a User Datagram Protocol broadcast packet which would reset the phone's clocks to the same time. Using this method, the phones were synchronized to within ten milliseconds for one hundred seconds at a time.

The phones show potential for future use in not only acoustic localization but many different types of distributed sensor networks.

Statement of Authorship

During the summer at Lincoln Laboratories, Scott worked on the Openmokos to install the Debian operating system. He also worked on creating the wireless network between the phones and other devices on the network. For this project, he implemented the signal processing and created the MATLAB GUI for testing.

Daniel worked at Lincoln Laboratories during the summer as an intern and helped create the wireless network between the phones. For the project he implemented the code to interface with the phone's audio drivers, as well as streaming that data to the central processing point. Additionally, he wrote the code for the time synchronization method.

For this project Muhammad, developed the method for synchronizing the phones based on a calibration chirp. Additionally he performed much of the testing and analysis of the data.

Acknowledgements

The authors would like to thank their advisors at Lincoln Laboratories, Glenn Schrader and Albert Reuther, as well as their advisors from WPI, Professor Ted Clancy and Professor George Heineman. Also they would like to thank the many people who helped them at Lincoln Laboratories, including Emily Anesta and Huy Nguyen.

Executive Summary

Acoustic localization is a system of determining the position of a sound source via an array of acoustic sensors. Historically, acoustic localization has been performed by hardware that was custom designed for the task. But if it were possible to use commercially available cell phones as acoustic sensors, then there could be a substantial savings in non-recurring engineering costs and a significant speed up in the time to realization of a system.

This project investigated the potential of using cell phones as sensors for the purpose of acoustic localization. Cell phones, in many ways, are well suited for this application. Cell phones already have microphones and wireless communication through phone networks. Additionally, many new cell phones have capable processors, wireless communication through Wi-Fi, and Global Positioning System (GPS) receivers.

This project attempted to use the Openmoko Freerunner as a sensor. The Freerunner is an open-source cell phone that runs its own version of Linux. The Freerunner has much of its hardware and software open to the public, as well as a large community of software developers. The phone has a 400 megahertz ARM processor, 128 megabytes of SDRAM, and is equipped with GPS and Wi-Fi.

The general setup that was constructed in this project can be seen in Figure 1. The setup had a local area network (LAN) consisting of two Openmoko phones and a laptop PC used as a central processing point. The two phones were connected through Wi-Fi to a wireless router with the central processing PC connected through an Ethernet cable. The phones were programmed to capture raw audio data using their built-in microphones and place the data into buffers that were

time stamped and queued to be sent to the central processing point. Once the data reached the central processing point, signal processing was performed to detect and locate a reference sound.

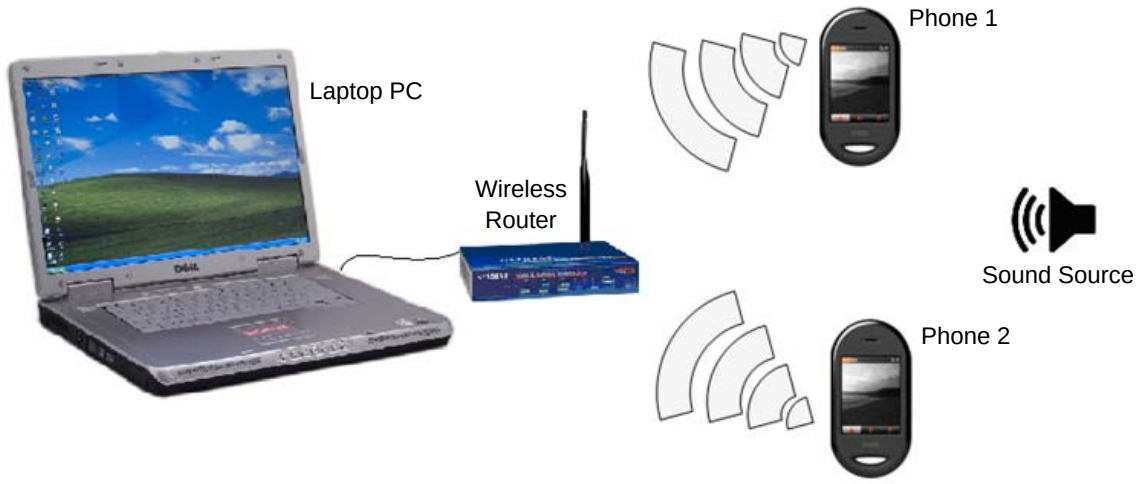

Figure 1 – Illustration of Hardware Setup for Acoustic Localization Using Cell Phones as Sensors and implementing a Central Processing Point. The Two Phones are Used as Sensors to Stream Captured Audio Data Wirelessly Through a Wireless Router to a Central Processing PC (Modified from [*Dell Inspiron*], [*FWG114pv2*], and [*Openmoko Image*, 2009]).

In this project, acoustic localization was attempted using a time difference of arrival approach. By knowing when a sound arrives at each phone and the geometric locations of these phones, a hyperbola of possible sound source locations can be drawn between the phones' microphones based on a known speed of sound. If another phone is added, three hyperbolas can be drawn, one for each combination of two phones. The intersection of these hyperbolas indicates the location of the sound source on a two-dimensional plane.

This method hinges on the ability of the phones to precisely be able to tell when a sound occurred. The speed of sound at sea level can be approximated as 340 meters per second, meaning each millisecond of error will result in *at least* 0.17 meters difference between the actual location of the sound source and the resulting hyperbola. Because of the large amount of

error that can occur because of poor time synchronization between phones, much of the investigation in this project took the form of implementing various means for time synchronization as well as measuring and reducing resulting errors.

One accomplishment of this project was achieving the necessary signal processing to detect a reference signal (in this project a chirp) when supplied with raw audio data in soft real-time. The detection methods were needed to accurately and quickly find the time difference at which the phones reported capturing the sound of the reference signal. This signal processing was used as a tool to test how well different time synchronization methods worked.

Different methods were explored to achieve precise time synchronization between the phones. One method would be to combine the high precision of GPS Pulse Per Second (PPS) signals with that of Network Time Protocol (NTP). PPS signals can be accurate to within 100 nanoseconds 99% of the time. NTP can use these signals to discipline the phones' clocks to account for drift, jitter, and variations from the clock's specified frequency. This method, however, cannot be used with the Openmokos as the PPS signals provided by the Openmokos' GPS chips are not physically connected to any other hardware. However, future explorations of this method may include contacting the manufacturer of the phone and convincing them to make future versions of the phone with their hardware to be slightly modified for increased abilities.

One of the most successful methods of time synchronization was using a User Datagram Protocol (UDP) broadcast signal. A UDP broadcast signal was sent throughout the network, ensuring that each phone would receive it at the same time. This method consisted of having the phones in a listening mode that would reset the phones clock to a predetermined time when the broadcast is detected. This method was found to be precise to within ten milliseconds for several minutes at a time.

Results from this method however, showed problems that needed to be solved if this or similar approaches to time synchronization were to be implemented. One problem was that the times between the phones would drift at a rate of approximately fifty microseconds per second, presumably caused by one of the phones system clock being slightly faster than the other. As time went on, the time synchronization between the phones became worse. The drift showed the need to periodically resynchronize the time between the phones. Test results for time synchronization versus elapsed time of the test can be seen in Figure 2.

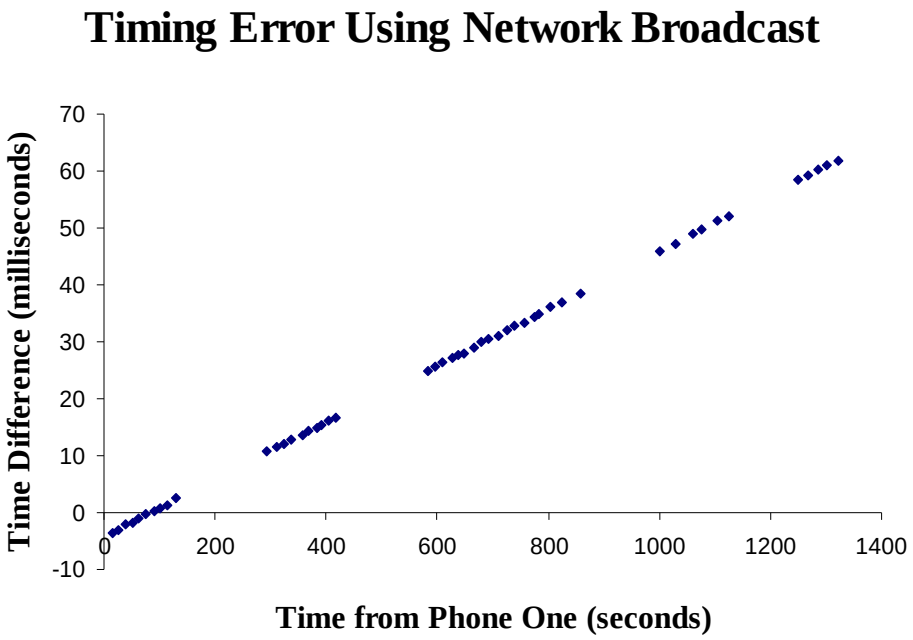

Figure 2 – Time Differences of Chirp Detections Between Two Phones that Used an UDP Method for Initial Time Synchronization. Initial Time Synchronization Occurs at Time Zero (Synchronization Error of Approximately). The Relative Time Between the Phones' Drifts Thereafter is at a Rate of Approximately 50 Microseconds/Second.

Other variations of this method were implemented, such as applying a moving average when interpreting the time stamps and adjusting when the time was read, which had limited success in improving the precision of the time synchronization between the two phones. All of these tests

however, have shown the capabilities of the Openmoko Freerunner and how it may have use in future applications. This piece of equipment has shown promise as a very capable device.

Table of Contents

1. Introduction

Developing custom hardware for deployment is an expensive and time consuming process. If the hardware also needs to be small, portable, and low power, then the expense and time to realization can be amplified even more. But what if there were already a hardware platform that met many of the requirements for mobile deployment? A cellular phone has already been engineered to be portable and conserve power.

Select modern cell phones have a very capable processor, wireless capability – both through 802.11 Wi-Fi and through cell phone networks – a GPS system, and a microphone. Additionally, with the development of Google Android and the iPhone it is becoming easier to develop software for them. To develop custom hardware and software that could perform to the level of these cell phones might take months, and the process would involve a substantial non-recurring engineering cost.

The goal of this project was to demonstrate the use of such modern cell phones as mobile sensors and create a network between them for the purpose of implementing an acoustic localization application. This project entailed producing a basic network that could stream data from phone sensors to a central processing point for application related processing. The project also focused on determining the sufficiency of using cell phones as part of a wireless sensor network.

Acoustic localization is a system of determining the position of a sound source via an array of acoustic sensors. Traditionally, acoustic localization has been performed by hardware that was custom built. Real-time analysis of audio streams to detect sound patterns takes a considerable amount of processing power which has historically been ideally suited for custom hardware. So

why build anything else? This project takes a different approach toward the problem, by using cell phones as acoustics sensors in order to show the benefits of using cell phones as sensors, as well as the detriments and limitations of the platform. Technical aspects that were explored included implementing a time synchronization methods between devices, process scheduling, and signal processing for detection of a reference signal.

2. Background

To understand the scope and technical details of the project, it is necessary to have a knowledge of the prior art in the field. First, the motivation of the project should be fully understood so that the project goals can be clearly laid out. Second, the prior art behind sensor networks and acoustic localization is necessary for understanding. Third, the mathematical background for acoustic localization will help with the understanding of the programming. Forth, knowing the capabilities and limitations of cell phones as mobile sensors will help with the design of systems. Finally, the ability for phones to network already exists, so knowing the capabilities of these networks will help to define the applications that can be developed.

2.1. Motivation

The main objective behind this project was to design a wireless sensor network using a commercially available cell phone to perform acoustic localization. There are three main benefits to using cell phones over custom hardware for acoustic localization. The first is a potential reduction in hardware costs. The second is a substantial reduction in non-recurring engineering costs. The third is a smaller time to realization for system prototypes. These benefits are discussed below.

Custom hardware for acoustic localization would require – at a minimum – some sort of processing device, a wireless communication method, a microphone for each sensor, and a power source for each sensor. Additionally, the system may require Global Positioning System (GPS) capabilities for location data or for time synchronization across the network, and/or memory – such as flash or RAM- to store data if the memory is not included with the processing device. There are also costs associated with manufacturing the sensors – such as producing custom

printed circuit boards. Each of these factors is already included in the cost of a cell phone, but the cell phone has the added benefit of being mass produced, so the overall cost is reduced due to economies of scale. While it cannot be said for certain whether a cell phone would be cheaper per sensor than custom hardware (that would depend on the requirements of the system), it should not be significantly more expensive.

The largest savings from using cell phones would not come from equipment costs, but rather the non-recurring engineering cost for development of the system. According to a report by the Bureau of Labor Statistics in 2008, the average hourly salary for an electrical engineer was $41.04. A computer software engineer for systems software commands an average hourly wage of $45.44. If a small team of three engineers (two electrical engineers, and one software engineer) working full-time, took twelve weeks to develop and test a system, $61, 209.60 would be spent on wages alone. This figure does not include the substantial over head of the project (such as insurance, an office, taxes, and utilities).

Implementation time is certainly another beneficial factor in using cell phones for acoustic localization. A lot of functionality is already available in a cell phone such as communication methods through Wi-Fi and phone networks, microphone drivers, GPS drivers, and an operating system. Readily available software development tools make it easy for a researcher to implement a program.

2.2. Mobile Hardware/Software Platforms

Not all cell phones are created with the ability for end users to develop applications. Recently, however, there has been a trend to allow more freedom with phones. There are many phones and platforms which may be enticing for future use as mobile sensors – such as the Nokia N900 or Google's Android operating system.

2.2.1. Openmoko

Openmoko is a project geared toward creating open source mobile phones. By default, the phone runs Openmoko Linux which is a software stack built on top of the Linux kernel. Additionally, Openmoko hopes to provide even more customization options to the end user by attempting to provide open source hardware. Both the schematics for the electronics as well as the CAD files for the phone are available from Openmoko.

The first Openmoko product, named Neo1973, was introduced in 2007 according to Openmoko's website. Shortly thereafter, the Neo1973 was followed by the Neo FreeRunner, released in June 2008, as an enhanced edition of Neo1973. Some of the key features of the Neo FreeRunner include a 400 megahertz ARM9 processor, wireless capability through 802.11g Wi-Fi, communication through GSM 2G cell phone networks, a microphone, a speaker, and GPS capabilities. Other features include 480x640 pixel touch screen display, 128 megabyte SDRAM, and 256 megabyte flash memory. Unfortunately, due to the economic recession, Openmoko recently announced the discontinuation of future phone development, but they plan to continue to support the Neo FreeRunner.

Being based on the Linux kernel, it was not long before community members provided different distributions of the operating system for the Openmoko. Some of the more notable distributions include scaled down versions of Gentoo, Debian/GNU, and Slackware Linux. [*OpenmokoWiki*]

2.2.2. Nokia N900

Termed by Nokia as a "mobile computer," Nokia N900 is a new addition to the open-source cell phone market, scheduled to be released in October 2009. The phone has an ARM Cortex-A8 600 megahertz processor with 256 megabytes of RAM. Similar to the Openmoko, the Nokia

N900 runs a Linux-based operating system. Fast wireless broadband is one of its unique characteristics; it is equipped with 3.5G and WLAN connectivity. It also provides Wi-Fi connections, a five megapixel digital camera, a 3.5 inch touch-sensitive display, and a GPS system. One of the particularly interesting features is the PowerVR SGX graphics core made by Imagination Technologies. According to the official website of Imagination Technologies, this graphics core can be used for general purpose computing. [*N900*, 2009]

2.2.3. Android Operating System

In 2008, Google released the system development kit for the Android Operating System to the public. Made available under the Apache license, Android provides users with an operating system, middleware libraries, and several applications. Using Google-developed Java libraries, developers can write applications for phones that support Android. Android gives users a large amount of portability across mobile phones. [*Android Website*, 2009]

While Google has not developed any hardware for the Android, there are now several phones on the market which support it (including the Openmoko). Also, Android offers high level access to devices features, such as GPS and wireless chipsets. Android might become a viable platform if a project needs to run the same applications across a wide variety of phones. [*Android Website*, 2009]

2.3. Sensor Network Deployment

Wireless sensor networks are composed of numerous sensor nodes where each node is capable of monitoring particular characteristics of the surrounding environment. Many challenges exist within the field such as detecting, monitoring, and analyzing data. Akyildiz et al.

[2002] performed a survey of some of the potential applications of wireless sensor networks. They explored applications in environmental, health care, home, commercial, and military areas.

Regarding military applications, Akyildiz and his team investigated several specific uses of networks. Sensors could be used for monitoring friendly forces and equipment. Terrain can be monitored for enemy troop movement. Guidance systems for missiles could be enhanced with information from nearby sensors. For homeland defense, sensor networks can be used to monitor for nuclear, biological, or chemical attacks.

Hardware constraints are an important consideration, according to Akyildiz. Each sensor device requires at a minimum a sensor, a processor, a transceiver, and a power unit. Additionally a sensor device may incorporate hardware for location finding (such as GPS), power generation, or movement of the sensor. A cell phone would only lack power generation and movement capabilities – although these functions could easily be incorporated into a design.

2.3.1. Recent Uses of Acoustic Localization

Acoustic localization is a technique for determining the position of a sound source using an array of microphones. Acoustic localization has already been implemented for military applications. In 2004, Ft. Belvoir Defense Technical Information Center published a paper discussing the success of detecting mortar and rocket fire in Iraq using acoustic localization. The US Army even recognized the sensors as one of the top ten inventions of 2004 [*Top Ten Inventions*, 2005]. Gunshot detection with acoustic localization has been deployed commercially for law enforcement purposes. The ShotSpotter is one such design. After being deployed, one city reported a thirty percent drop in complaints of shots being fired [*ShotSpotter Testimonials*, 2009].

2.4. Networking

Many options are available for networking sensors. Sensors can exist with a central base station which controls all communications, or sensors can route traffic through other sensors. By using cell phones as sensors, there is also the choice of using cell phone networks or 802.11 Wi-Fi.

2.4.1. Ad hoc Networks versus Managed Networks

An ad hoc network is one where there is no central location through which all traffic is routed. Packets of information can be sent through other nodes dynamically as they enter and leave the network. In wireless sensor networks, an ad hoc network can be desirable because there is no point with which all sensors need to communicate. If the sensor network is deployed over a large geographical area, the power required to transmit the data to other nodes can be reduced. An ad hoc network can also make the system more robust. If one node is destroyed, the network can still function. [Perkins, 2001]

In contrast to the ad hoc network, one can have a managed network where all traffic is routed through a central point. This routing method drastically reduces the complexity required for setting up the network and routing traffic, at the cost of needing central points with which to connect. [Perkins, 2001]

2.4.2. Wi-Fi Capability versus Phone Networks

If one is building custom hardware, there are many possible methods for communication between sensors. With phones however, one is limited to two practical methods. One is through 802.11 Wi-Fi and the other is communication through the phone networks.

Wi-Fi communication is defined by standards designated by the IEEE. With the 802.11g standard there is a maximum net bit rate of fifty-four megabits per second. However, 802.11 was never meant to send data over long distances (farther than a few hundred feet). This limitation means that sensors cannot be spread very far apart. Furthermore, with 802.11, development, expansion, and maintenance of the network would be left to the end user. [*IEEE 802.11*, 2008]

Using the phone networks for communication in a sensor network has many advantages. Phone networks already provide a pre-existing infrastructure which spans over a large geographical area. On the other hand, the throughput of a phone network suffers. AT&T claims to offer the fastest 3G wireless network in the country, but only boasts typical speeds of 220-320 kilobits per second [*3G*, 2009]. However, this value is quickly increasing. Sprint has plans to launch a 4G network which they claim will have average speeds of three to six megabits per second [*Sprint 4G*, 2009].

Ultimately, the decision for which network to use will depend on the application. Faster speeds will require a Wi-Fi connection, while longer range sensors will need to use the phone networks. Also, for longer range applications a phone may use Wi-Fi to connect to a nearby wired network in order to move the data further.

2.5. Time Synchronization

There are many possible methods for synchronizing time between wireless devices. A few of them are presented below.

2.5.1. Global Positioning System

The Global Positioning System (GPS) is a global navigation satellite system. GPS receivers provide three dimensional location data and time data using satellite broadcast signals. The

signals can only be received there is a relatively direct line of sight between the satellite and the GPS chip. The signals can pass through clouds, glass, and plastics, but not underground or within enclosed areas [Xu, 2007]. GPS receivers, such as the ones within an Openmoko Freerunner, are capable of creating a Pulse Per Second (PPS) signal with a precision of less than one hundred nanoseconds ninety-nine percent of the time [GPS datasheet, 2009]. Unfortunately, the PPS signals provided by the Openmoko's GPS hardware come out of a pin which is not connected to any of the other hardware. Therefore, the Openmoko is unable to use the GPS for accurate time synchronization of its clocks.

Time synchronization plays a significant role in acoustic localization. The location of a sound source is determined solely by the difference in time of arrival between each sensor. In order to have an accurate time of arrival difference, it is vital to have the system clocks of the sensor devices synchronized with each other. Even one millisecond of time difference would cause at least 0.17 meters *minimum* of error in the distance between the sound source and the sensors.

Another aspect to consider is using the GPS receivers for positional data. By already having GPS receivers available in many cell phones, they could potentially add even more functionality to the application. The current accuracies for positional data from GPS are roughly 2.0-2.5 meters Circular Error Probability (CEP) and 3.0-5.0 meters Spherical Error Probability (SEP). CEP is defined as the radius of a horizontal circle, centered at the GPS receiver's true position, containing 50% of the fixes. Similar to CEP, SEP is the radius of the sphere, centered at the true position, containing 50% of the fixes [GPS datasheet, 2009]. If errors in time synchronization and other areas can be limited, these errors could be tolerated within the scope of an acoustic localization application.

2.5.2. User Datagram Protocol

User Datagram Protocol (UDP) could be used as a method to synchronize times between sensors in acoustic localization. UDP messages are part of the Internet Protocol suite used for data transfers without connections [Clark, 2003]. The devices in the network can listen to the for a message from a sender, and as soon as they receive it, a predetermined time could be set on the system clocks of the devices. This method is crude, and does not allow for resynchronization should the clocks of the sensors drift over time. But, for some applications, resetting the clock times every few days may provide enough accuracy to use this method. Network Time Protocol (discussed below) works by sending multiple UDP messages.

2.5.3. Network Time Protocol

Network Time protocol (NTP) is part of the Internet Protocol suite and is used to synchronize the clocks of computers to a time reference Normally, NTP works by connecting to several NTP servers to get the time. NTP will then take into account factors such as latency between each server to slowly correct the system's clock. However, NTP also has the ability to use a PPS signal as a time "server". In this situation, NTP will slowly correct the system clock based on many received PPS signals. Over time, it will correct for clock drift, jitter, and variations from the clock's specified frequencies. NTP could be particularly useful in sensor networks as it has the ability to synchronize a computer's system clock using GPS. The GPS hardware produces a PPS signal based on GPS data received from satellites. NTP could be used to synchronize the system clocks using GPS as its time source. In the application of acoustic localization, using NTP with GPS could be particularly useful as GPS has the potential to provide great amount of accuracy and precision. [Römer *et al.*, 2005]

2.5.4. Reference Broadcast Synchronization

Reference Broadcast Synchronization (RBS) is one of the synchronization protocols that requires receivers to use the broadcast messages to compare the clocks with each other. This method is different from the other traditional sender-receiver methods in which the sender transmits the timing information and the receiver synchronizes. RBS uses receiver to receiver synchronization. The idea is similar to that of the previously described UDP, in that, the sender sends a packet to the receivers without any timing information and the receivers use their local clocks to record when the packet was received. Subsequently, the receivers determine their relative offset with one another by exchanging their timing information. The calculation of this offset is based on when the receivers receive the reference signal. [Holger, 2005]

There are numerous advantages associated with using RBS as the time synchronization method. RBS can be used in a topology as simple as one sender and two receivers or it could be made complex with more than two receivers and possibly more than one broadcast. This structure can be suitable for many different applications in sensor networks, especially acoustic localization. RBS is a very energy conservative method for time synchronization as it participates in post-facto synchronization. In this mode, the receivers do not synchronize with each other until an event of interest takes place. [Holger, 2005]

2.5.5. Radio Signals

Using radio signals to synchronize times between the devices is a more traditional approach to time synchronization. Custom equipment outfitted with low-cost radio receivers could be used to utilize the signal for precise timing. Radio time signals can be used by the custom equipment to provide a precise time reference. They are synchronized with precise atomic clocks. The local date and time information is broadcasted periodically and the time data are transmitted as pulse-

width encoded data signals. Radio signals travel at the speed of light; therefore, if two receiving sites have a difference in distance between their position and the location of the sending site of twenty meters, there will be approximately 66.7 nanoseconds difference in the time of arrival of the signal. Obviously, phones are not built with the hardware required for this synchronization, so it cannot be used without significant modification. [Webster, 1999]

2.6. Theory

There are two main components to acoustic localization: detecting the sound and triangulating the position of the source. For simplicity, one can use a sound that can be easily detected such as a chirp. One method for detecting a chirp is called cross-correlation.

2.6.1. Chirp Characteristics

Chirps are signals where the instantaneous frequency increases over time. Chirps are easy to detect because of their unique frequency characteristics. They are commonly used in sonar or radar applications. A chirp has many useful properties, such as having an equal amount of power over a range of frequencies, not occurring commonly in nature, and having the ability to be detected in the presence of significant background noise. An example of a chirp can be seen in the Figure below.

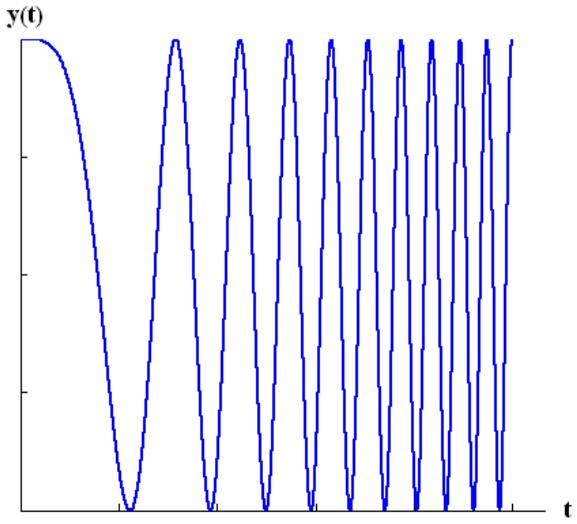

Figure 2.1 - Sample Chirp Signal where the *t* Axis is Representative of Time.

2.6.2. Cross-Correlation and Autocorrelation

Cross-correlation is a method to determine the resemblance between two random-valued functions. It is very similar to convolution in signal processing. In convolution, a function is reversed, shifted and then multiplied by another function and the results are summed. Whereas in cross-correlation, the function is only shifted and multiplied by another function with the results being summed [Lasko, 2002]. This concept is often useful when searching a long signal for a shorter known feature embedded within that signal. Cross-correlation for two real-valued discrete-time functions f[m] and g[m] is defined as [Elali, 2004]:

$$x[n] = \sum_{m=-\infty}^{\infty} f[m]g[n+m]$$

In the above equation, *f* and *g* are two separate functions defined in terms of *m*. Function *g* is shifted by the value *n*. Function *x* represents the cross-correlated result of the two functions *f* and *g* in terms of the shift or lag *n*.

Autocorrelation is defined as a signal that is cross-correlated with itself. Autocorrelation of a chirp will result in a sinc function with a peak at a lag of zero, which represents no time shift being applied to either function. An example of autocorrelation of a chirp can be seen in Figure 2.2.

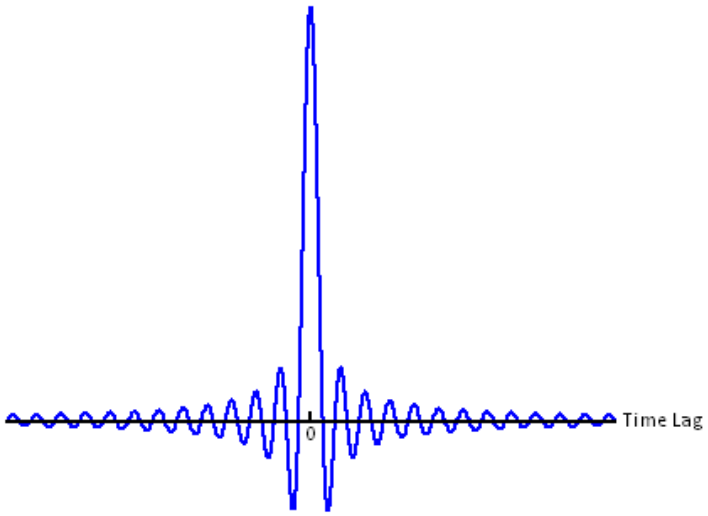

Figure 2.2: Resulting Waveform from Autocorrelation of a Chirp.

Autocorrelation of a chirp generates a well defined peak, as shown above. Based on this concept, when cross-correlating a predefined chirp with another signal, if a similar chirp is embedded within that signal, a resulting peak will indicate when it occurred.

2.6.3. Cross-Correlation in the Frequency Domain

Implementing cross-correlation becomes increasingly inefficient as the length of the signals that are being cross-correlated grows. For longer signals, it is generally faster to perform the calculations in the frequency domain. The steps for performing correlation in the frequency domain are as follows for signals of length M and L.

1) Pad both signals so that they are length M+L-1.

2) Perform the Fast Fourier Transform (FFT) of both signals.

3) Find the complex-conjugate of one of the FFTs.

4) Multiply the signals point-wise.

5) Take the Inverse Fast Fourier Transform (IFFT) of the resulting signal.

When one signal is much longer than the other and cannot be stored in memory, then the longer signal is usually separated into smaller, more manageable sections. There is a problem associated with separating the signal, however [Madisetti and Williams, 1998].

Using the above method, and breaking the signal into smaller sections will result in more points than is desired. This discrepancy occurs because there is overlap between the sections which is not being accounted for. It may seem like a simple solution to pad only one signal, however this would not work because the end of the correlation signal would be added to the beginning of the correlation signal in a process called cyclic correlation, which is also not desirable [Madisetti and Williams, 1998].

One possible way to account for the overlap between sections and avoid the aliasing caused by cyclic correlation is to implement an overlap-save method for correlation. The key to understanding the overlap-save method is to realize that when signals of length M and L are convolved in the frequency domain, the first M-1 points will be affected by cyclic correlation, but all other points are valid. Therefore, one can pad each section of the longer signal with M-1 points from the previous section of the longer signal (or zeros if it is the first section). Then the steps are performed as outlined above. After the IFFT has been done, the first M-1 points of the result are discarded and the result is concatenated with previous results [Madisetti and Williams, 1998].

The overlap-save method will greatly increase efficiency for long signals. In practice, the FFT of the shorter signal and its complex conjugate can be pre-computed. The method results in performing one FFT which is O(Nlog(N)), N complex multiplications, and one IFFT which is also O(Nlog(N)) multiplications. These complexities can be compared to correlation in the time domain which is $O(N^2)$ [Madisetti and Williams, 1998].

2.6.4. Hyperbolic Solution

Figure 2.3 shows a setup of an acoustic sensor array with two microphones and one sound source.

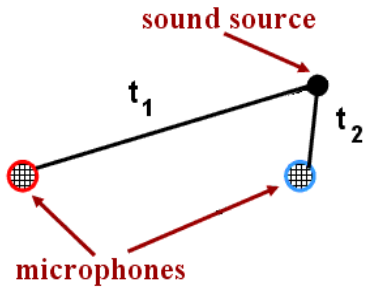

Figure 2.3: Acoustic Localization with Two Sensors (Modified from [Birchfield and Gillmor]).

In Figure 2.3, the variables t_1 and t_2 represent the amount of time it takes for the sound to travel from the sound source to each of the sensors. The sensors, however, will only be able to report times at which the sound was detected, meaning that the only variable that can be determined from these sensors is time difference between t_1 and t_2. If it is assumed that the speed of sound is constant, then the time of sound arrival difference will be proportional to the difference in distance from the sound source to each of the sensors. The time and distance differences can be shown mathematically with the following equation where Δd is the distance difference, Δt is the time difference, and v_s is the speed of sound:

$$\Delta d = \Delta t \times v_s .$$

The only other variable that is needed to be known is the distance between each of the sensors. In this derivation, a two-dimensional Cartesian coordinate system will be constructed by using the sensors' positions (F1 and F2), modeling the system in one plane. The x-axis is defined as the line that intersects each of the sensors. The y-axis is perpendicular to the x-axis and intersects the x-axis at the mid-point between the sensors. The distance from one sensor to the y-axis will be represented as *c*. The point at which the sound source lies will be defined as *P*, with variable coordinates of *x* and *y*. The variables defined for this derivation can be seen in Figure 2.4.

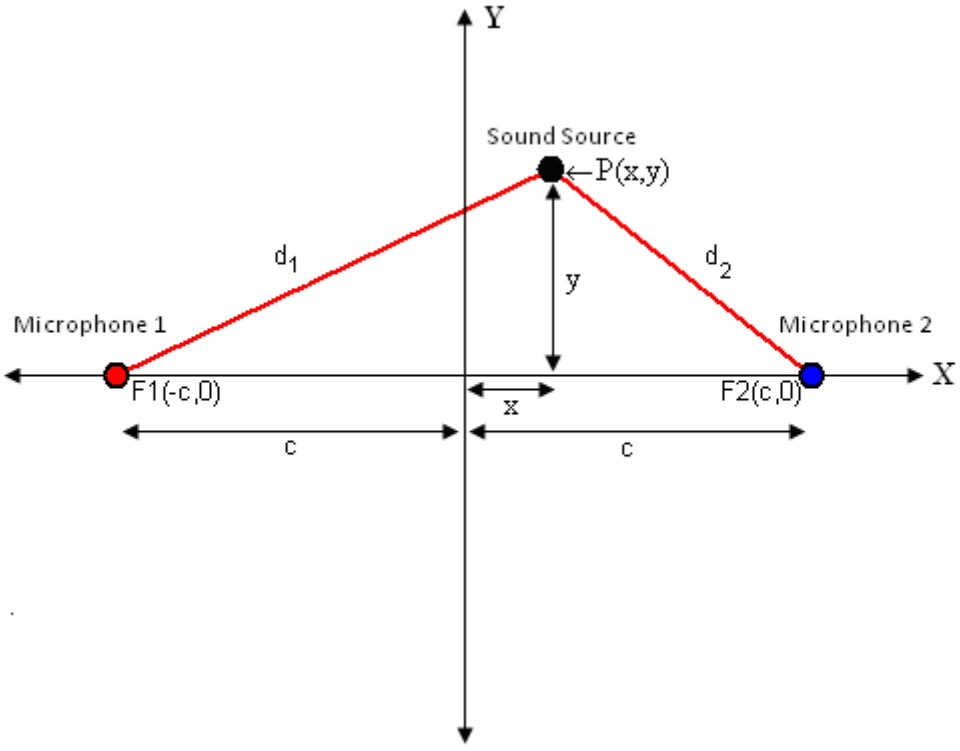

Figure 2.4 – Visual Aid for Mathematical Derivation of Acoustic Localization Calculation.

Figure 2.2 illustrates the acoustic localization scenario, where d_1 and d_2 are the distances from each of the sensors to all possible points *P*. The difference in distances between *P* and the two fixed focal points is Δd , written as:

$$\Delta d = d_2 - d_1.$$

Note that although this derivation is setup in a manner so that the sound source is closer to sensor two, if the sound source was actually located closer to sensor one (resulting in a negative Δd) the equations would still apply. Applying Pythagorean's theorem to d_1 and d_2 gives:

$$d_2^2 = (x-c)^2 + y^2$$
$$d_1^2 = (x+c)^2 + y^2.$$

By solving for d_1 and d_2 and substituting into the previous equation for Δd , the following relation results:

$$\sqrt{(x-c)^2 + y^2} - \sqrt{(x+c)^2 + y^2} = \Delta d.$$

As shown by Casey in 1893, the equation shown above is that of a hyperbola. The equation can also be derived from the definition of a hyperbola where the distance difference from two focal points (in this case the sensors) is equal for all points on the hyperbola. Casey uses the methods described below to manipulate the above equation into the form of a simplified hyperbolic equation.

Define a new variable *a* which is equal to $\Delta d/2$:

$$\sqrt{(x-c)^2 + y^2} - \sqrt{(x+c)^2 + y^2} = \Delta d = 2a.$$

Rearranging the terms and completing the square gives:

$$x^2(c^2 - a^2) - a^2y^2 = a^2(c^2 - a^2)$$.

Dividing both sides by $a^2(c^2 - a^2)$ gives:

$$\frac{x^2}{a^2} - \frac{y^2}{c^2 - a^2} = 1$$.

Remember that *c* is a measured distance, and *a* is derived from the time difference of arrival as reviewed in the equations below:

$$\Delta d = \Delta t \times v_s, a = \frac{\Delta d}{2}$$.

A variable *b* can be defined such that,

$$b^2 = c^2 - a^2$$.

Therefore, the equation of a hyperbola is given by:

$$\frac{x^2}{a^2} - \frac{y^2}{b^2} = 1$$

As a result, time delay between a pair of sensors combined with the known distance from one to the other transforms into a solution which is a hyperbola in 2D space. A visual representation of two sensors and a sound source with the hyperbolic solution is shown in Figure 2.5.

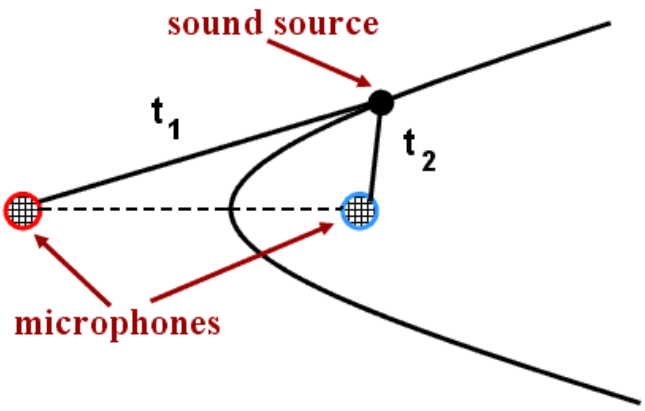

Figure 2.5 - Acoustic Localization with Two Sensors and Derived Hyperbolic Curve (Modified from [Birchfield and Gillmor]).

The sound source can reside anywhere on the hyperbola. In order to solve for a unique point of origin in the plane, another sensor is needed and must be placed so that it is not collinear with the other two sensors. This setup allows for a hyperbolic solution between each pair of microphones. The intersection of the hyperbolas will be the location of the sound source. A visual representation of this scenario can be seen in Figure 2.6.

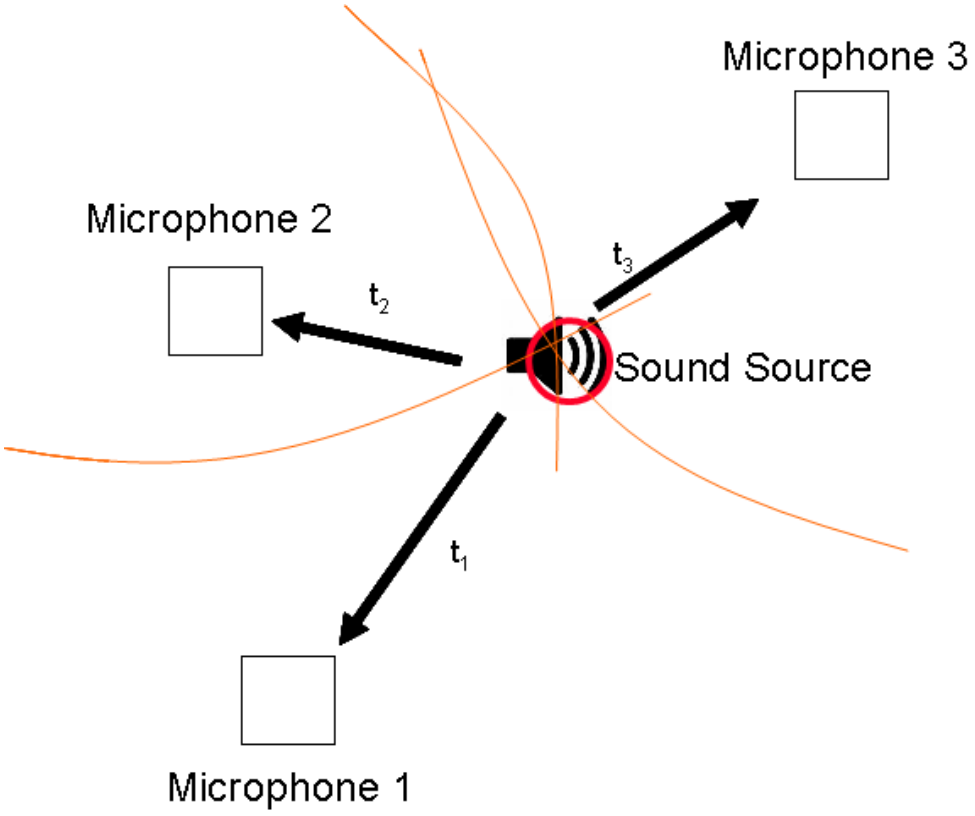

Figure 2.6 - Visual Representation of an Acoustic Localization System with Three Sensors.

The variables in Figure 2.6 (t_1, t_2, and t_3) represent the time it takes for the sound to travel from the sound source to each of the sensors. In this scenario, there are three sensors being used which implies that three hyperbolas will be created, one from each of the time of sound arrival differences.

2.6.5. Error Analysis of Acoustic Localization

The greatest source of error acoustic localization in this project was from the error in time synchronization between the phones. Figure 2.7 shows an example of error that can be generated from a ten millisecond time difference with two phones twenty meters apart and the sound source located equidistant from both the phones.

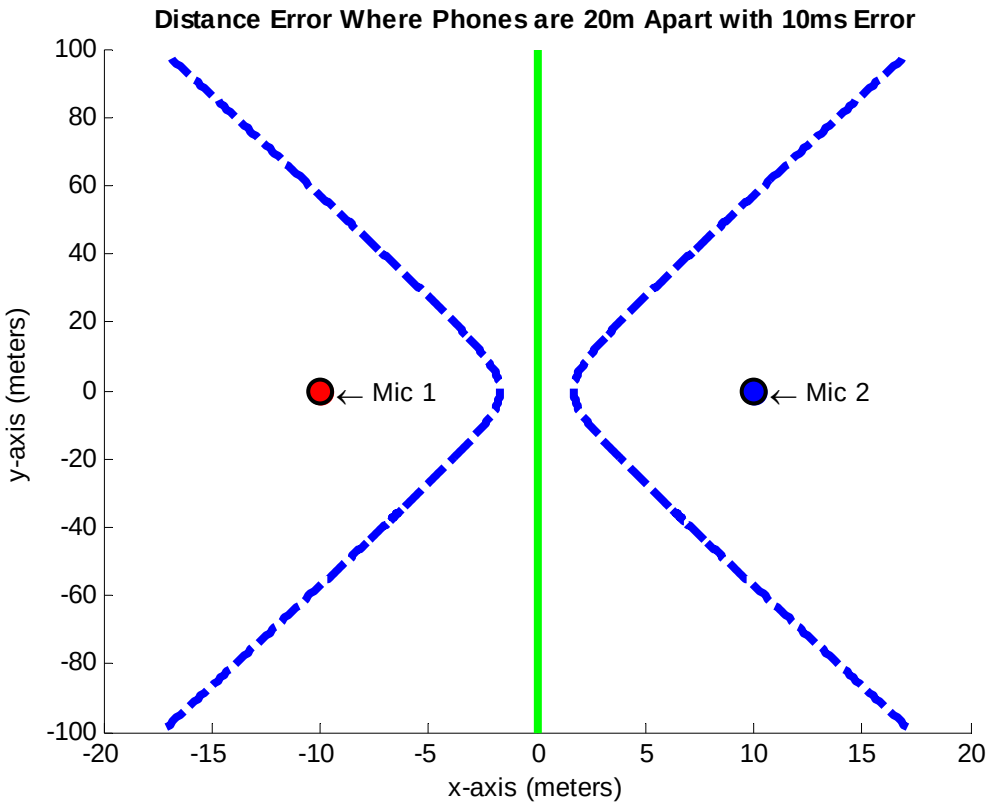

Figure 2.7 – Error Analysis of Acoustic Localization with Two Sensors Placed 20 Meters Apart. Solid Green Line Shows "True" Sound Location Possibilities for the Case When the Sound Arrives Simultaneously at Both Phones. Dashed Blue Lines Show the Solutions with 10 Milliseconds of Error.

The green line in Figure 2.7 represents the correct line that the sound source was located on. The two dashed blue lines represent the resulting line from plus or minus ten milliseconds of error in time of arrival. The minimal amount of error, also the error of *a* from the hyperbolic calculations, would be 1.7 meters. It can be seen that if the sound source was located one hundred meters in the y direction, the error would be a minimum of seventeen meters.

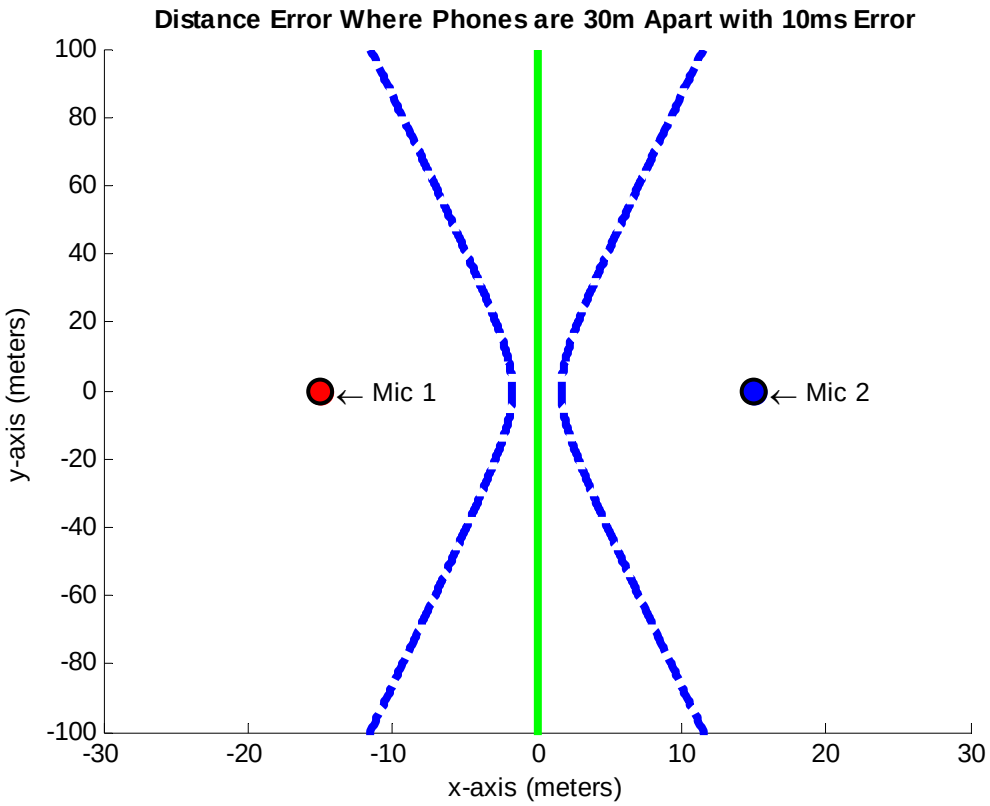

Figure 2.8 – Error Analysis of Acoustic Localization with Two Sensors Placed 30 Meters Apart. Solid Green Line Shows "True" Sound Location Possibilities for the Case When the Sound Arrives Simultaneously at Both Phones. Dashed Blue Lines Show the Solutions with 10 Milliseconds of Error.

In Figure 2.8, the sensors are placed thirty meters apart from one another. The green line represents the correct line that the sound source was located on. The two dashed blue lines represent the resulting line from plus or minus ten milliseconds of error in time of arrival. The minimal amount of error would be 1.7 meters. Notice that the distance that the sensors are away from one another does not affect the minimal amount of error. It can be seen that if the sound source was located one hundred meters in the y direction, the error would be a minimum of twelve meters. This Figure shows how if the sensors are placed farther apart from one another the error lines will diverge at a slower rate.

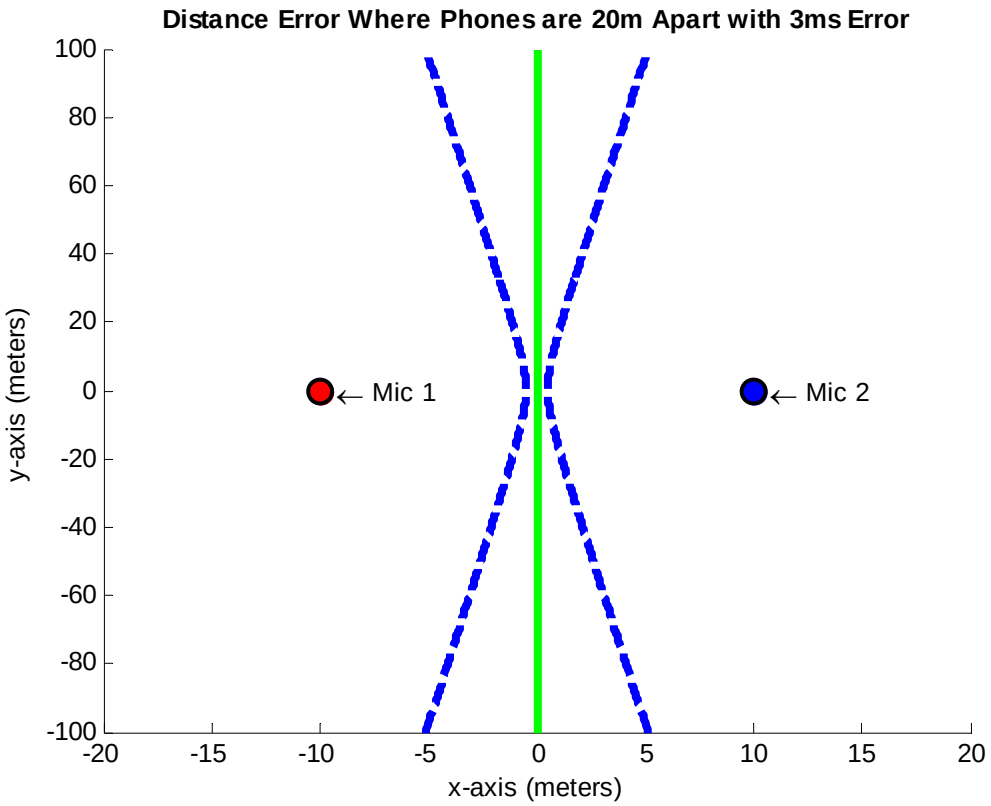

Figure 2.9 – Error Analysis of Acoustic Localization with Two Sensors Placed 20 Meters Apart with 3 Milliseconds of Error. . Solid Green Line Shows "True" Sound Location Possibilities for the Case When the Sound Arrives Simultaneously at Both Phones. Dashed Blue Lines Show the Solutions with 3 Milliseconds of Error.

In Figure 2.9, the sensors are placed twenty meters apart from one another. The green line represents the correct line that the sound source was located on. The two dashed blue lines represent the resulting line from plus or minus three milliseconds of error in time of arrival. The minimal amount of error would be 0.51 meters. It can be seen that if the sound source was located one hundred meters in the y direction, the would be a minimum of five meters. This shows how if the error could be consistently reduced to three milliseconds, how the minimal error would reduce as well as how the error lines will diverge at a slower rate.

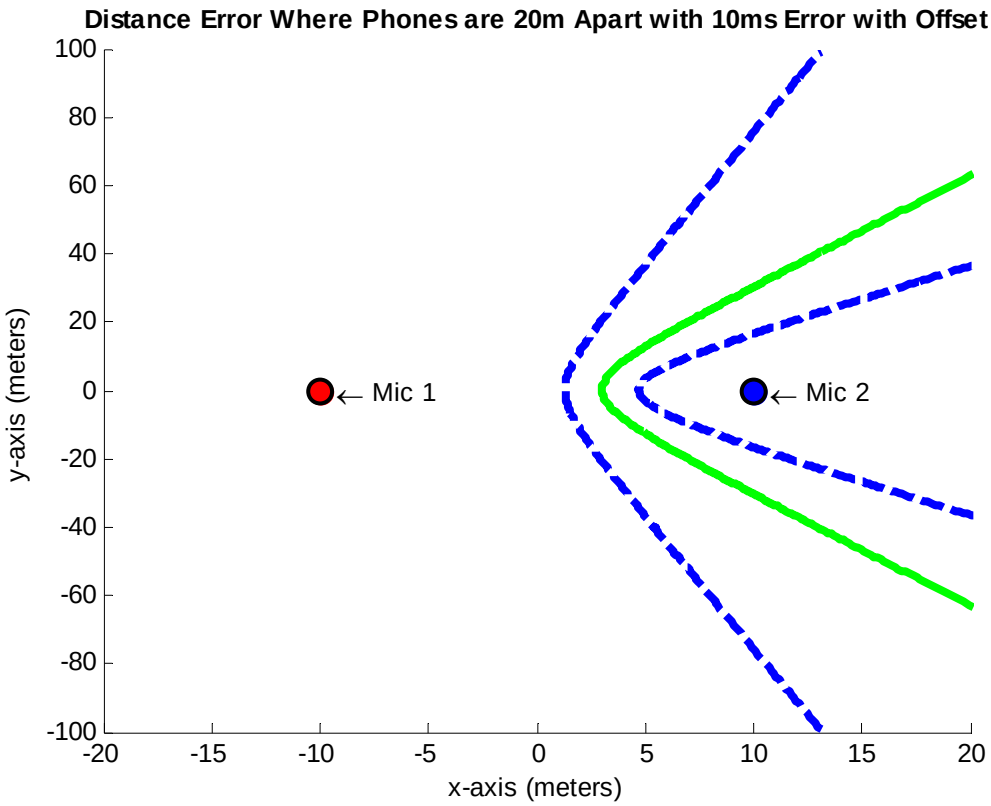

Figure 2.10 – Error Analysis of Acoustic Localization with Two Sensors Placed 20 Meters Apart and the Sound Source Closer to Microphone Two. The Solid Green Line Shows "True" Sound Location Possibilities. Dashed Blue Lines Show the Solutions with 10 Milliseconds of Error.

In Figure 2.10, the sensors are placed twenty meters apart from one another. The green line represents the correct line that the sound source was located on. In this case, the sound source is not equidistant from the microphones. The two dashed blue lines represent the resulting line from ten milliseconds of error in time of arrival. The minimal amount of error would be 1.7 meters. Notice how the error lines are diverging at different rates. If microphone two's clock was ten milliseconds ahead than microphone one's, then there would be less error than if microphone one's clock was ahead. The location of the sound source with respect to the phones and which phone's clock is ahead factors greatly into the positional error for acoustic localization.

2.6.6. Speed of Sound Variance

The speed of sound is a factor within the acoustic localization process. The speed of sound varies based on many air qualities – mostly on air temperature. The equation used to calculate the speed of sound based on temperature is as follows:

$$V_{sound} \approx 331.4 + 0.6 \cdot T_c \left[m / s \right].$$

In the above equation T_c is the temperature of the air in Celsius and V_{sound} is the speed of sound in meters per second [Howard and Angus, 2006]. When considering accuracy of the localization calculation, temperature could potentially be an important factor to take into account.

3. GPS Positional Data

This project considered using GPS for positional data since it is already implemented on the phone. The acoustic localization platform could become more mobile by having positional data readily available to the system for use in distance calculations. This test explores the precision of the GPS positional data from the Openmokos.

3.1. Methods

In order to test the precision of the GPS modules, the Openmokos were placed side by side where the long edge of each of the phones touched and the microphones each facing the same direction. The phones were then placed outside on a flat surface two feet from the ground. There was no cloud coverage, giving the phones a clear line of sight. If the GPS worked perfectly, the data recorded would be precise to within the distance between the phones GPS receivers (a few centimeters). The two phones then each executed a python script called getgps which is readily available on the Openmoko Wiki. This script was modified to wait for a signal from a GPS satellite and when the phones first received a signal the script would begin to poll the GPS every ten seconds and record the data to a file on the phone. The test was run for approximately ten minutes. The data were then further analyzed to calculate an average distance between the two phones using the Haversine formula [Bretaña, 1987].

3.2. Results

Figures 3.1 and 3.2 show a histogram of the latitudinal data gathered from the phones using the methods described above. Figures 3.3 and 3.4 are histograms of the gathered longitudinal

data from the test. Table 3.1 contains the mean, median, minimum, maximum, and standard deviation for each of the measured parameters.

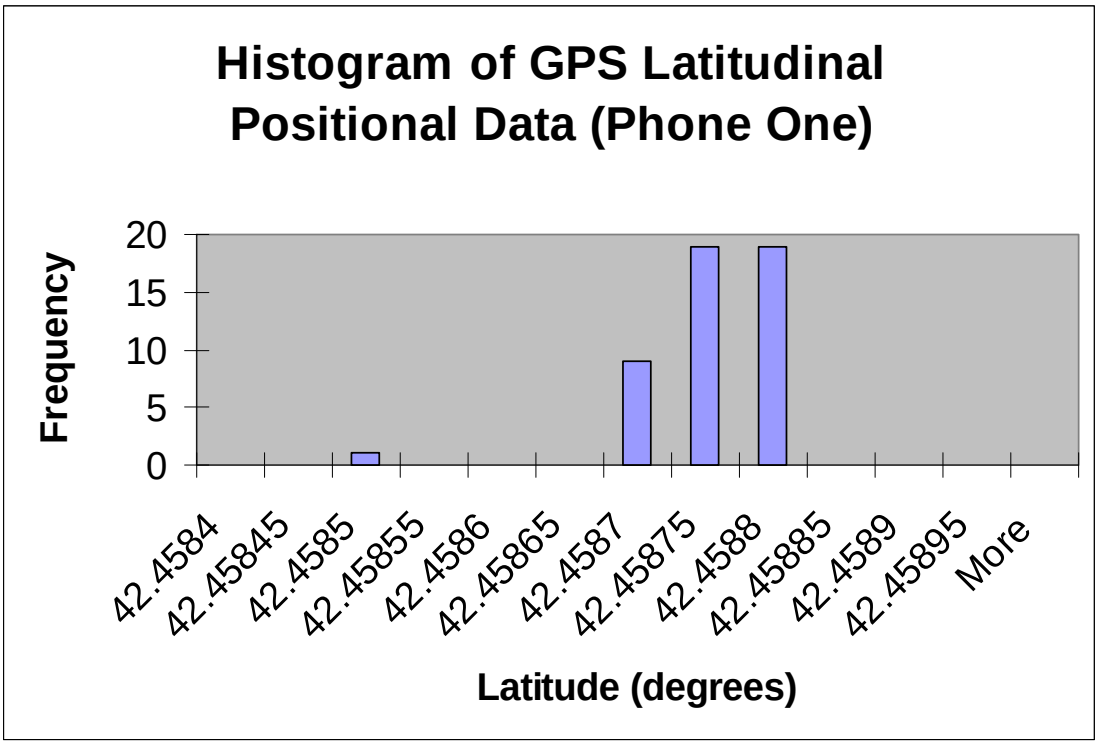

Figure 3.1 – Histogram of Latitudinal GPS Data Measured Using an Openmoko (Phone One) Over a Period of Ten Minutes.

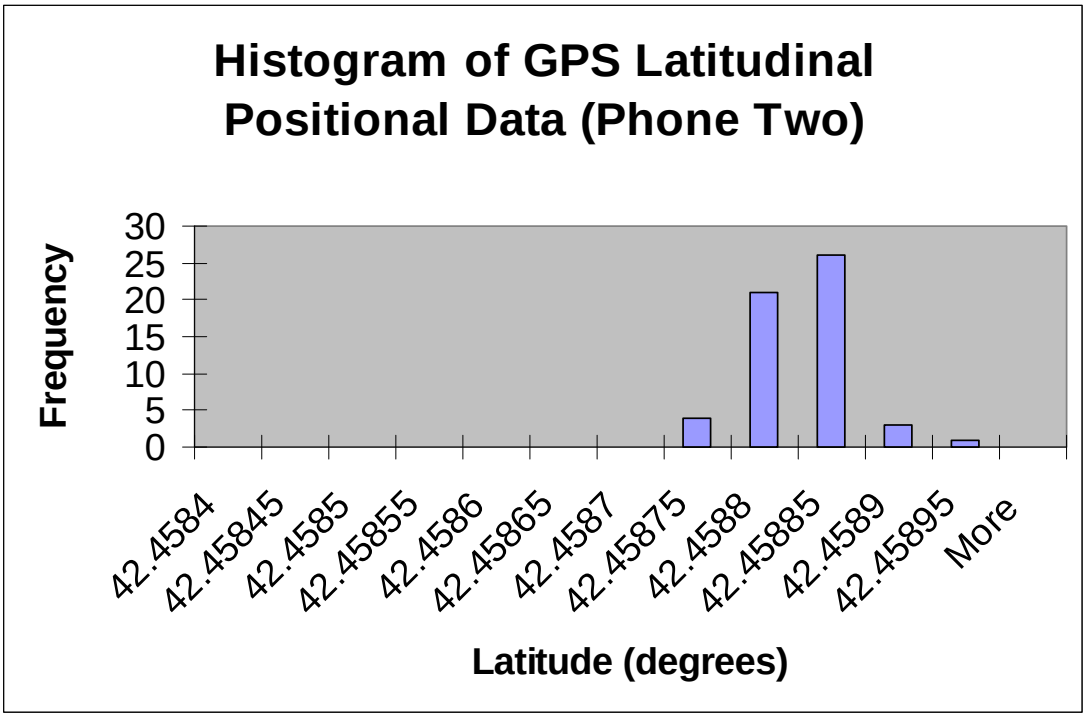

Figure 3.2 – Histogram of Latitudinal GPS Data Measured Using an Openmoko (Phone Two) Over a Period of Ten Minutes.

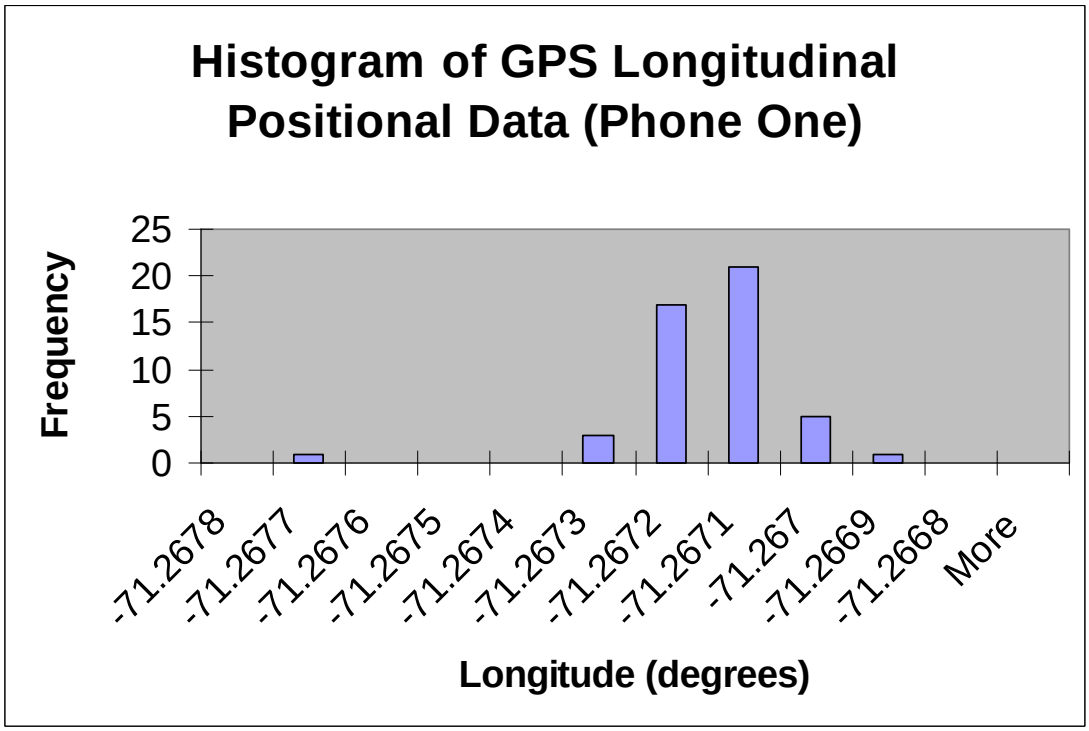

Figure 3.3 – Histogram of Longitudinal GPS Data Measured Using an Openmoko (Phone One) Over a Period of Ten Minutes.

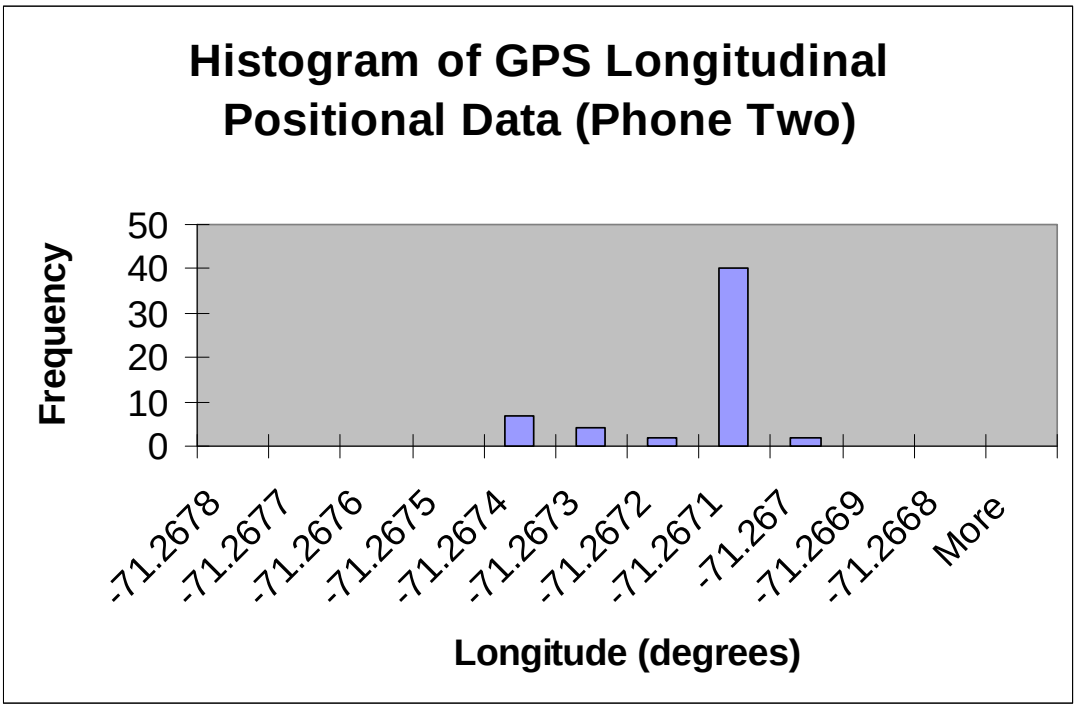

Figure 3.4 – Histogram of Longitudinal GPS Data Measured Using an Openmoko (Phone Two) Over a Period of Ten Minutes.

Table 3.1 - Mean, Median, Minimum, Maximum, and Standard Deviation of the GPS Positional Data Measured Using Openmoko Cell Phones Over a Period of Ten Minutes.

	Latitude (Phone One)	Latitude (Phone Two)	Longitude (Phone One)	Longitude (Phone Two)
Average (degrees)	42.45873158	42.45880242	-71.26719263	-71.26719871
Minimum (degrees)	42.458473	42.458739	-71.267753	-71.267466
Maximum (degrees)	42.4588	42.458916	-71.266997	-71.26709
Standard Deviation (degrees)	5.20216E-05	3.86622E-05	0.000112699	0.000107609

The distance between the average latitudes and longitudes between the phones was calculated using the Haversine formula:

Distance between the average latitudes and longitudes: 7.887 meters

3.3. Discussion

According to Table 3.1 the average values for latitude and longitude between the phones are the same within four decimal places. Similar accuracies can be seen in the minimum and maximum values of the latitudinal and longitudinal positions. However, the distance between the average latitudes and longitudes was 7.887 meters. Depending on the application, these precisions may be acceptable, but there are orders of magnitude of difference in precision when compared to measurements with statically placed phones at known positions (assuming the distance between the phones could be measured to within a few centimeters).

4. Setup for Streaming Audio Data

In this project, acoustic localization was divided into five distinct tasks:

1) The microphones of the cell phones listen to their surrounding area.

2) The sensory data are streamed over a wireless network.

3) At a central processing point, the data are analyzed to detect an event.

4) Detected events, which were caused by the same source, are associated with one another at the central processing point.

5) The central processing point calculates the location based on the time difference of the events.

An overview of the process division can be seen in Figure 4.1.

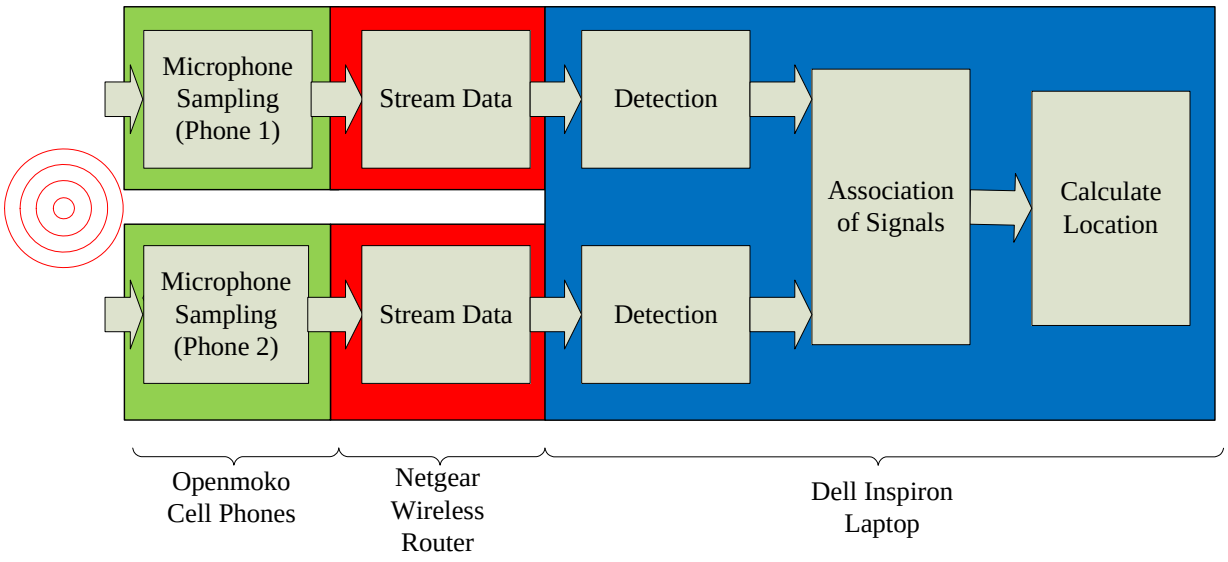

Figure 4.1 – Block Diagram of Acoustic Localization Method Using Cell Phones as Sensors.

The reasoning behind the decisions that lead to this architecture is described in the following sections.

4.1. Cell Phones

The design for this project's sensor network is based on two principles. First, custom hardware is expensive and time consuming to produce. Second, with each passing year, more off the shelf technology is becoming available that may be able to achieve the same needed task. These benefits are why cell phones are used as sensors in this project. Cell phones have many hardware capabilities that are needed to be part of a sensor network. For this project two Openmoko Freerunners were used. Both Freerunners are hardware version A6. Debian GNU/Linux with version 2.6.29 of the kernel was installed on the Freerunners due to the extensive package repository available for the ARM architecture.

4.2. Central Processing Point

The Openmoko was used because it was readily available to the researchers. Unfortunately, the Freerunner lacks floating point hardware, which makes the sound detection algorithm much harder to implement. Because this system is only for demonstration purposes, the data were streamed from the phone to a central processing point (via an 802.11g wireless LAN) to be analyzed.

The central processing point was a Dell Inspiron 700m laptop. This computer has a dual core 2.10 gigahertz Pentium M processor and 2.0 gigabytes of RAM. The laptop was running Microsoft Windows XP 32-Bit Edition Version 5.1 (Build 2006.xpsp_2_gdr.090206-1233: Service Pack 2). The signal processing was done on this computer in MATLAB version 7.8.0.347 (R2009a).

4.3. Signal Processing

The audio signal that was used for testing in the project was a chirp. Chirps can be easily detected by using a cross-correlation method as described in the background. The chirp that was used in this project was generated in MATLAB using the chirp function. In general, a chirp should be as long as possible, so for this project a 100 milliseconds long chirp was used, which was about the length of time that could be processed reliably before MATLAB would run out of memory. The chirp was a discrete-time signal created to be played at a rate of 44,100 hertz. Chirps are easier to detect when the bandwidth of the chirp is large. Thus, the frequencies of the chirp used in this project varied linearly from 100 hertz to 20,000 hertz, which was the maximum bandwidth that could be produced with the speaker used in this project. Figure 4.2 shows the raw audio data that one of the Openmokos recorded while a chirp was playing during the User Datagram Protocol time synchronization test (the methods for the test will be explained later in this paper). The units for this graph are in ADC values, which correspond to the sound pressure level at the microphone. The Figure also shows the signal that occurred when the chirp was cross-correlated with the reference chirp. The units of the cross-correlation are in ADC Values^2, which corresponds to how well the two signals are correlated. The final graph in the Figure shows the spectrogram of the chirp which was generated from the raw audio data in MATLAB using a Hamming window of length 256 and 250 overlapping segments to produce 50% overlap between segments.

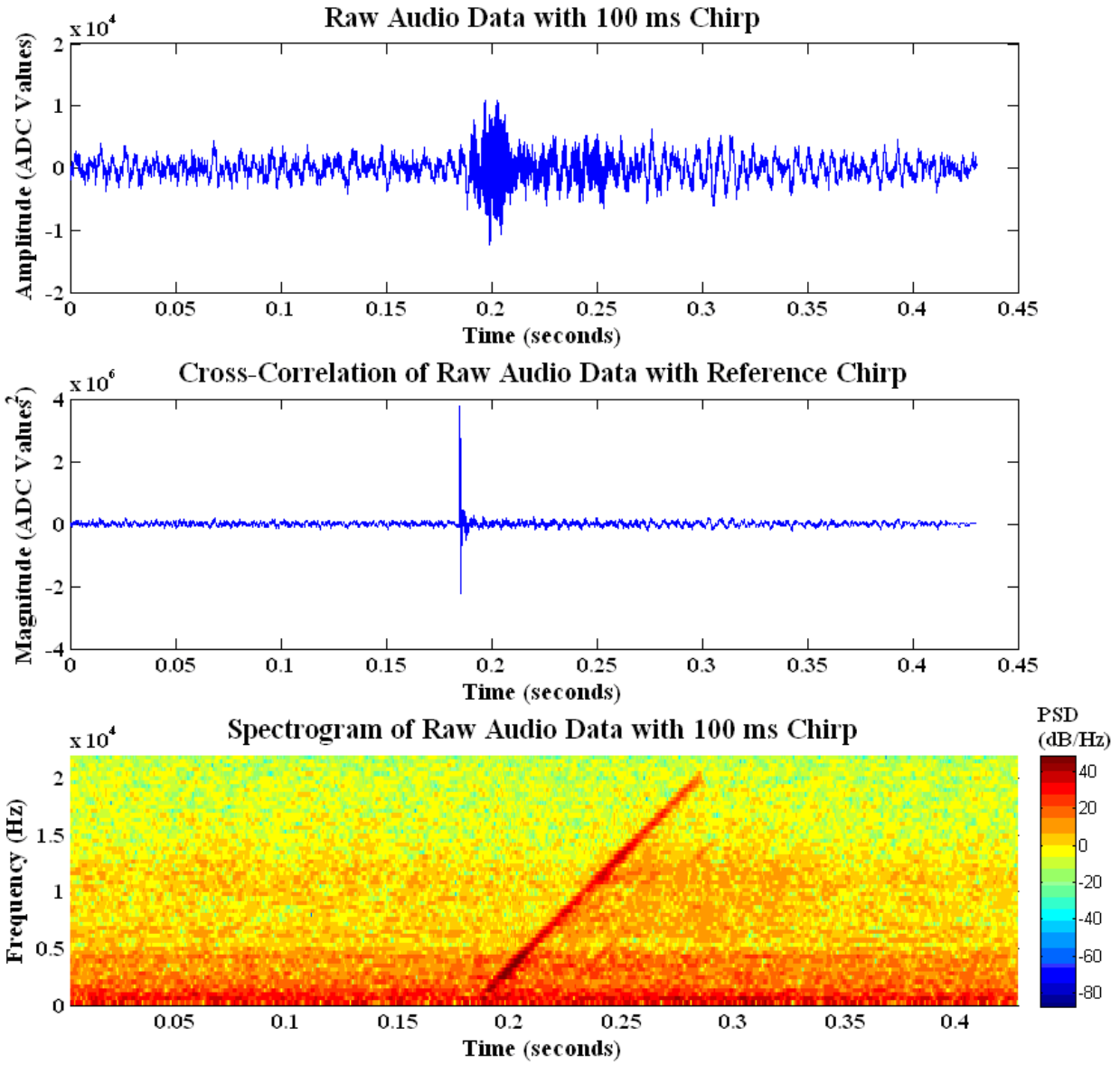

Figure 4.2 – Raw Audio Data of a Chirp (Top) Compared To Cross-Correlation of Audio Data and Reference Chirp (Middle) and Spectrogram of Raw Audio Data (Bottom). The Chirp Occurs Just Prior to 0.2 Seconds.

The spike (detection) within the cross-correlation plot does not appear to correspond with the time occurrence of the chirp in the raw audio data, however, as one can see from the spectrogram, there is a large amount of interference in the lower frequencies due to the ambient noise in the testing environment (most likely the ventilation in the room). The lower frequencies of the chirp are not readily identified in the top plot because of this noise.

Based on this chirp and similar chirps, a threshold value of 10^6 ADC Values^2 was used. Any correlation that produced a value greater than this would be considered a peak. Looking back at Figure 4.2, this value is clearly much greater than the noise values. The cross-correlation in this case only produced three points that are greater than the threshold. Three points above the threshold give a 68.0 microsecond window where the start of the chirp could lie. Other cross-correlations tended to produce three to six points that passed the threshold.

4.4. Streaming Data Over a Wireless Network

Due to the ease of implementation, throughput, and cost, this project utilized a Wi-Fi wireless network instead of streaming the data over a phone network. The wireless network between the phones was set up using a Netgear FWG 114P wireless router with firmware version 2 using WPA2 certification.

The sound card on the phone is capable of producing 32 bits of data for each audio sample at a rate of 96,000 samples per second [*WM8753 Datasheet*, 2008]. To transfer audio data at this rate would require speeds of three megabits per second, which falls well within the range of 802.11g's maximum throughput of 56 megabits per second.

4.4.1. Method of Recording and Sending Audio Data

The phones' program to record and send audio data was written in C because of C's low level access to data and its speed. The method which it used to record and send audio data can be broken up into five steps as shown in Figure 4.3 and described below:

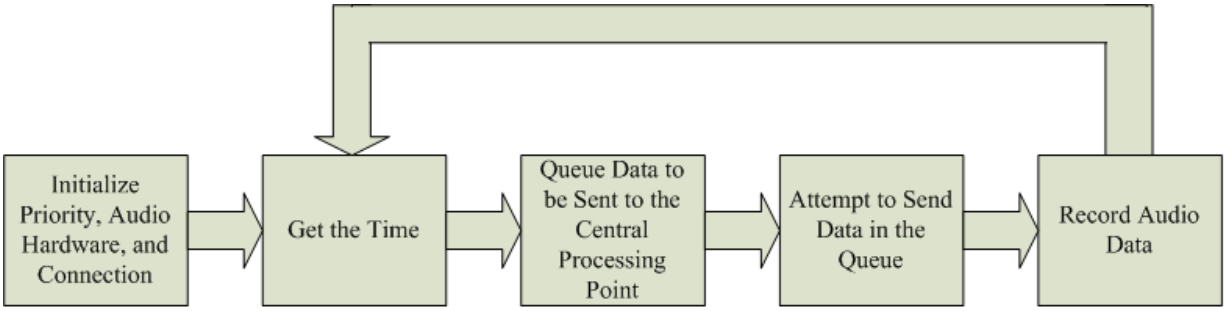

Figure 4.3 – Processing Diagram of Phone's Program to Record and Send Audio Data.

1) Initialize priority, audio hardware, and network connection.

The first step of the C program was to configure the priority and scheduling policy of Linux. Since other processes using the hardware could interfere with the data being collected, the sched_setscheduler function in the GNU C library was used to set the priority of this process to be higher than other processes. Additionally, the function was used to set the policy for scheduling to be first in first out. By doing this, it ensured that this program will be given the top priority and almost immediate access to the hardware whenever it needed to run.

Next, the C program configured the audio hardware. To configure the hardware, the program used the Advanced Linux Sound Architecture (ALSA) Application Programming Interface (API). The code used the ALSA API to set the hardware to record at a sampling rate of 44,100 hertz with 16 bits of data per sample with a single channel. Additionally, the hardware was set to record 1024 samples for each request to read, and the data were recorded in big-endian format so they can be directly transferred over the network without conversion.

After the audio hardware was configured, the program would open a simple TCP/IP socket and listen for a connection from the central processing point. When the MATLAB code on the laptop connected, MATLAB would tell the phone the size of its buffer. It is important to note that the buffer size of 1024 samples in the C program on the phone did not correspond with the number of samples MATLAB would read at a time; that is to say, the buffer size of the

MATLAB code was not the same as the buffer size of the C code. The reason for different sized buffers will be explained in the section on receiving data from MATLAB. After receiving MATLAB's buffer size, the code on the phone would enter into the main loop of its program.

2) Get the time from the system clock using the gettimeofday function

The first step in the main loop was to simply get the time of day. The time of day was read as two, unsigned, thirty-two bit values – the number of seconds since January 1, 1970 at midnight and the number of microseconds since the last second. The times the phones sent were not as important as the precision at which they sent them. That is to say, both phones could have sent times that were set in January of 1970 as long as the times for both were precise with respect to each other.

The time that was read may or may not be sent over the network. For ease of implementation, the time was only sent at the start of every MATLAB buffer, which, as mentioned earlier, did not correspond with the size of the audio buffer in the C code. In fact, the size of a MATLAB buffer was also not usually evenly divisible by the size of the C buffer. The C code would keep track of how many samples have been sent to MATLAB so far and send the time when the MATLAB buffer would start again. When the number of values fell in the middle of the phone's audio buffer, the code would first queue the audio data before the end of the MATLAB buffer, it would then correct the time it read by adding twenty-three microseconds per sample before the end of the buffer. It would then queue the rest of the data to be sent. Twenty-three microseconds corresponds to the inverse of the sampling frequency, the real value is closer to 22.6757. Twenty-three was used in order to avoid floating point calculations. This rounding resulted in approximately .3243 microseconds of error per data point. However, since the entire buffer was only 1024 samples long, in the worst case scenario (1023 samples before the MATLAB buffer

ends) there was only be 34.89 microseconds of error. If the time was not sent (because the C buffer was ending, but the MATLAB buffer was not), it was be discarded.

3) Start the A/D converter to record 1024 audio samples.

After the audio hardware has been configured correctly, all that was done to record audio was issueing a read command with a pointer to a memory location allocated for the audio data.

4) Queue the data to be sent over the network.

It was important that the system go through this programming loop as quickly as possible. If the program spent too much time attempting to send data over the network, then there would be a large gap of time where no data were being recorded. In an attempt to reduce the time a program may try to unsuccessfully send data over the network, and any time or audio data to be sent was put into a queue to be sent.

5) Attempt to send the first set of time or audio data in the queue, if successful move to the next set of data if not, go to step 2.

The queue operated by attempting to send the first set of data, but not sending it if sending the data would have resulted in the program waiting – that is, it sent the data in non-blocking mode. If the program would have ended up waiting for the send to complete, then it did not remove any data from the queue, and it went back to the second step. If the data could be sent without waiting, then the data were removed from the queue and sent, then the program would attempt to send the next items on the queue. It would continue this process until either the data could not be sent without waiting or there were no data left in the queue.

4.4.2. Method of Receiving and Processing Audio Data

The MATLAB code that read in and analyzed data from the phones can be broken down into six steps as shown in Figure 4.4 and described below:

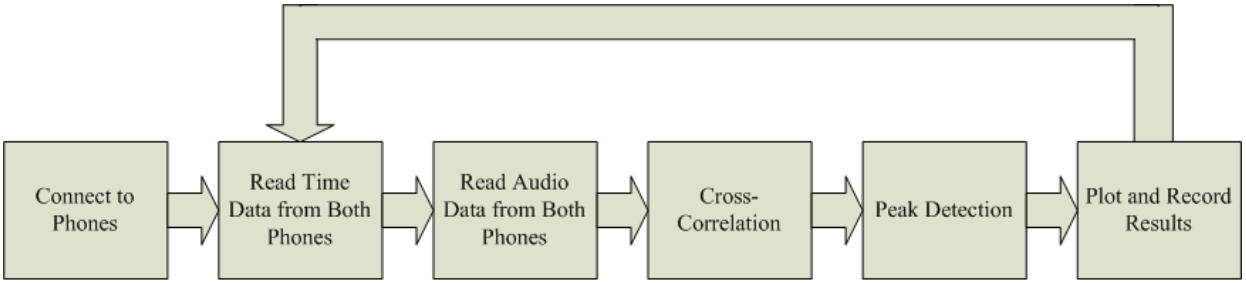

Figure 4.4 – Processing Diagram of MATLAB Code to Analyze Audio Data.

1) Make a connection to the phones and send the size of the read buffer in MATLAB.

MATLAB's first step was to determine the size of the read buffer. As mentioned before in the introduction, the length of the target chirp plus the length of the audio data being read minus one was a power of two so that the Fast Fourier Transforms (FFTs) would perform quickly. To make FFTs longer, the number of audio samples in an FFT was also chosen such that it would be four to eight times larger than the number of samples in the reference chirp. Additionally, the data needed to be read for multiple FFTs so that MATLAB did not waste too much time going back and forth to get more data. By default, the MATLAB code in this project would read in ten FFTs of data before it went to do processing (although this value could have been easily changed). A MATLAB buffer was defined in this project to be the amount of data that are read each time through the program loop (by default ten FFTs of data). It should be noted that this value was generally very large, and it was not generally evenly divisible by the size of an available buffer in the C code. For example, with a one-hundred millisecond chirp, ten FFTs of data, and a sampling rate of 44,100 samples per second, the MATLAB buffer contained 283,580 samples. Therefore, the C code had to keep track of the number of samples it had sent, and it only sent the time at the beginning of a MATLAB buffer.

MATLAB has native support for communication over TCP/IP networks, such as the one set up in this project. After computing the buffer size, MATLAB made a connection to each phone, sent the buffer size to both phones, and waited for data. Each connection was separate and data

was not be mixed between them. MATLAB had an additional TCP/IP buffer which stored values that had been sent, but not yet read into a MATLAB buffer. This buffer was set at the arbitrarily large value of three megabytes. If the system was performing in real-time, this buffer would never fill completely.

2) Read in the time data as four thirty-two bit unsigned integers.

The C code on each phone would send the time stamps as two values – one unsigned integer for the seconds and one for the microseconds. Since there were two phones, there were four values to read.

3) Read in the audio data as sixteen bit integers.

In this step, MATLAB would fill the audio buffer with enough data to fill its buffer.

4) Perform the cross-correlation.

Before the cross-correlation was performed, the data from the MATLAB buffer were parsed so that several cross-correlations could be performed at once. This step included sectioning off the data and overlapping the sections so that the overlap-save method described in the introduction would work. The cross-correlation was performed using MATLAB's xcorr function. The xcorr function performs the cross-correlation in the frequency domain, according to the source code of the xcorr function. Furthermore, xcorr implements additional optimizations such as using fftfilt when the length of the signals to be cross-correlated differ by a factor of ten or greater. Because of these optimizations, using the xcorr function resulted in better defined peaks than a manual implementation. This project's implementation did not take advantage of pre-computing of the FFT of the chirp, but since the system operated in real-time using xcorr there was no reason to rewrite the function. After the cross-correlation, the sections of the signals where circular correlation occurred were removed, and the results were concatenated.

5) Detect the peaks

Peak detection was done in MATLAB using the findpeaks function. In order to only define one value for a peak, a minimum peak distance of 8000 samples was set. This value corresponded to an 18.14 millisecond window around each peak where another peak would not be detected. If more than one sound occured in this time period, the system would only detect one of them for each phone, and it was not guaranteed to be the same peak that each phone detected. However, since this was a demonstration system, the sounds would never occur that close together.

Additional code checked to make sure that a peak did not exist on the very edge of a MATLAB buffer. If a peak occured within 15,000 samples of the end of the buffer, it was marked as having the potential to overlap. The data were saved and after the next MATLAB buffer was cross-correlated the code checked for peaks that are within 15,000 points of the start of the buffer. If there were peaks on both the start and end of the buffer, the code would only count the larger of the two peaks. Otherwise, the peak that occurred was saved and the program moved on. This constrained the system even more. If two peaks occurred within 30,000 samples of each other (approximately 680 milliseconds), then it was not guaranteed to detect both of them. However, the value of 15,000 was chosen only because it was large; this value could have been safely reduced to the largest number of points that could occur above the peak threshold and still be guaranteed to work.

6) Plot the results, and, if there was a peak, write the results out to a file.

No matter what was in the signal, the MATLAB code created three graphs for each phone. The first graph plotted the raw audio data from the phones over time. In order to limit the number of points that needed to be drawn, however, not every point was displayed. Instead, a

variable dictated the number of points to skip over before plotting the next point. By default the system would only plot every sixty-four points. The graph was not very accurate and would show aliased data, but it served to give a general idea of what noises were occurring.

The second graph consisted of the cross-correlation. Again, not every value was plotted. Instead, the code plotted the same number of points as the audio data plot. With cross-correlation one could easily miss the peak by skipping over that many points, however. Therefore, the code only plotted the maximum values for each buffer. The graphing was done by sorting the cross-correlation values in descending order and keeping track of the where the values occurred in time, and then the top values were resorted by index. The resulting plot did not correspond exactly to time, but was useful for debugging or a human operator.

The final graph consisted of the amount of data left in the TCP/IP buffer. For every MATLAB buffer, the TCP/IP buffer was plotted three times. The value was plotted right after the audio data were read, right after the signal was plotted, and right after the cross-correlation and peak detection occurred. This graph helped evaluate the real-time ability of the system. If the values were steadily increasing, then the system would eventually run out of memory and drop data. A screenshot of the graphical user interface with the plots is shown in Figure 4.5.

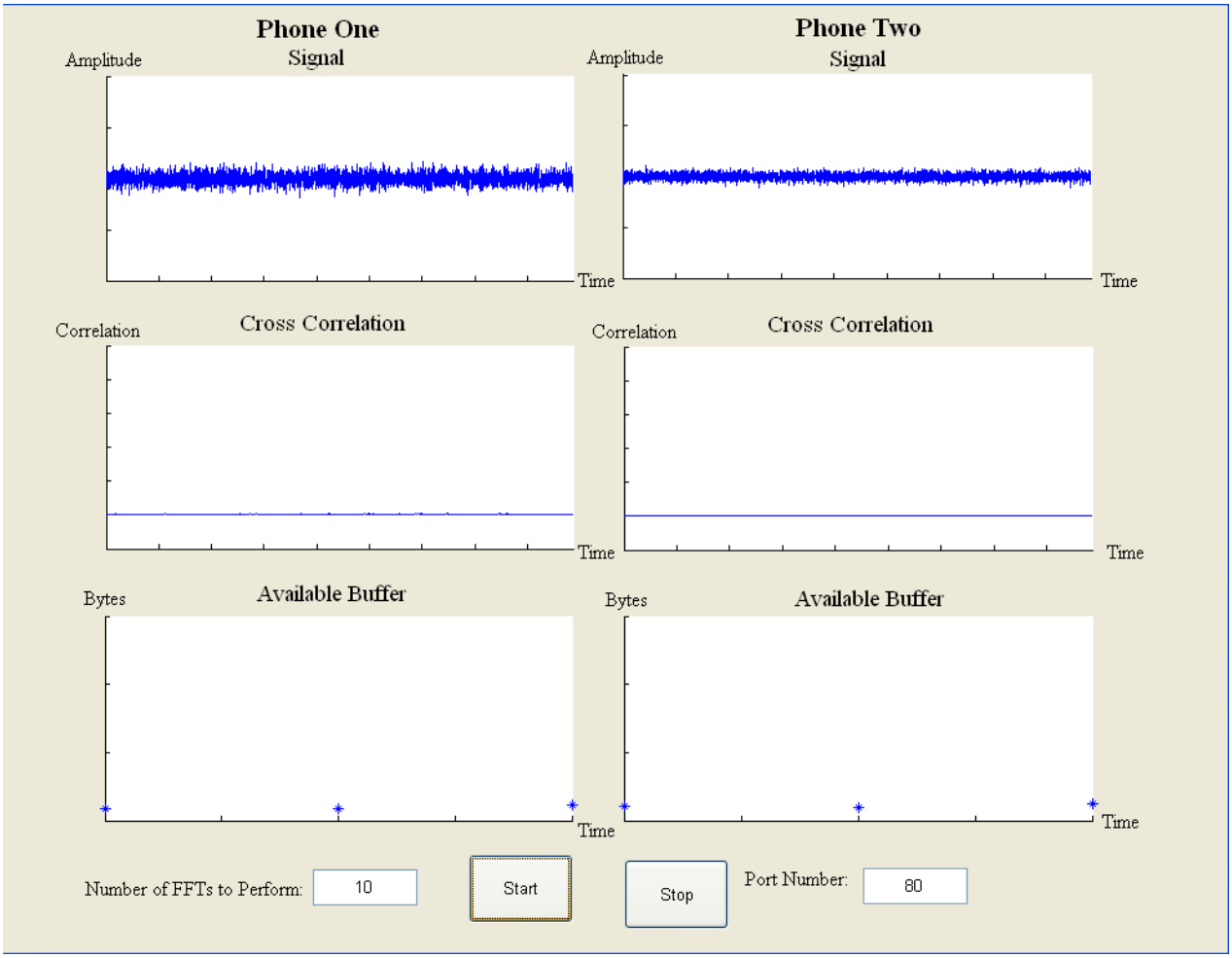

Figure 4.5 – Interface Showing Graphs of Raw Audio Data Signal, Cross-Correlation and Buffer Size in MATLAB. Left Column is for Phone One, the Right Column is for Phone Two. The Top Row is the Raw Audio Data. The Second Row Displays the Cross-Correlation of the Raw Audio Data. The Third Row Displays Available Memory in the MATLAB Buffer.

The graphs, although helpful, did not provide accurate information which could be analyzed later. That is why data were also recorded to files anytime a peak occurs. There were two files created for every peak. One file contained the raw audio data of the MATLAB buffer, in which the peak occurred, written to a file as a series of sixteen bit integers. The second file had three sets of data: the seconds at the beginning of the MATLAB buffer, written out as a thirty two bit unsigned integer; the microseconds at the beginning of the MATLAB buffer, written as a thirty

two bit unsigned integer; and the cross-correlation written out as a series of double precision floating-point numbers.

5. Read and Queue Execution Time Test

One of the most important steps to determining how well the Openmokos are able to synchronize their clocks is determining how precisely each phone is able to read its system clock. This section will describe the experiment used for determining how precisely the phones can read their clock.

5.1. Methods

For this test, the C code on the phones was modified to write the seconds and microseconds to a file whenever the time was read. The MATLAB code was unaltered, but no chirps were played; the central processing point served only as a location for the phones to send their data. This test was run for ten and a half minutes, resulting in 27040 time values for one phone and 27067 time values for the other (the discrepancy in data points is due to the programs being stopped manually at different times).

On each phone individually and for every time value that was read, except for the first time value, the previous time value was subtracted. Since the C code was reading 1024 samples between the times when the system clock was read, there should have been a 23.219 millisecond difference between each of the time values (plus a small amount of constant time for queuing and sending the data).

5.2. Results

Figure 5.1 shows the histogram of the resulting time differences that were read on the test for one phone while figure 5.2 shows the histogram of the resulting time differences that were read on the test for the other phone.

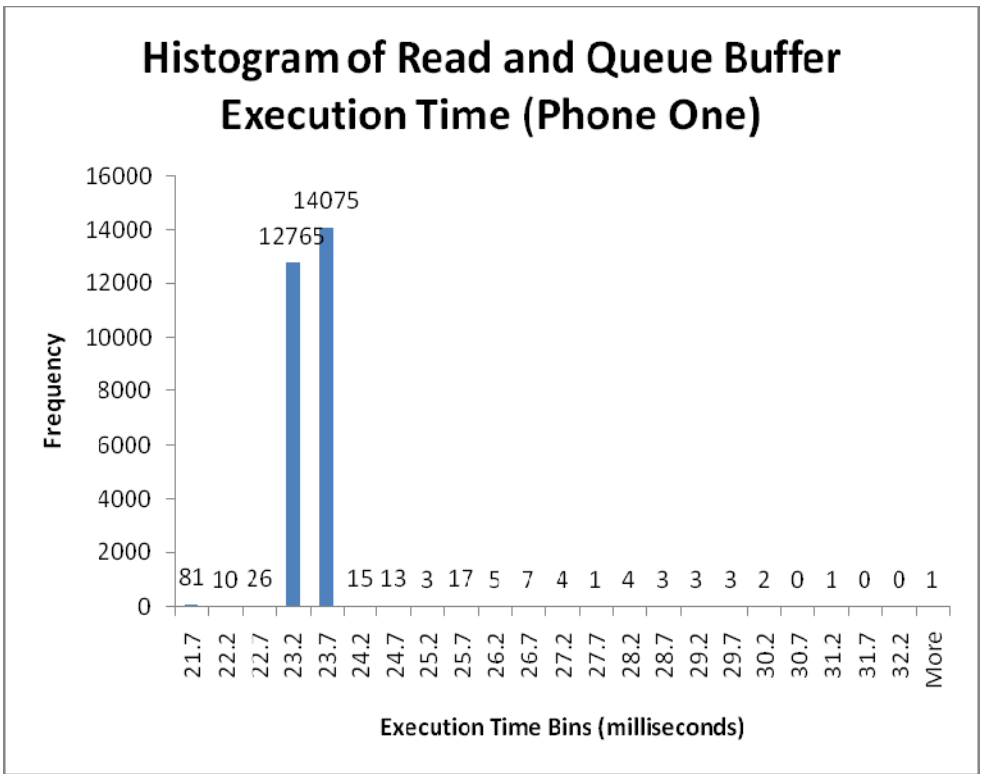

Figure 5.1 – Histogram of Time Difference Points Between Reading Audio Buffer on Phone One.

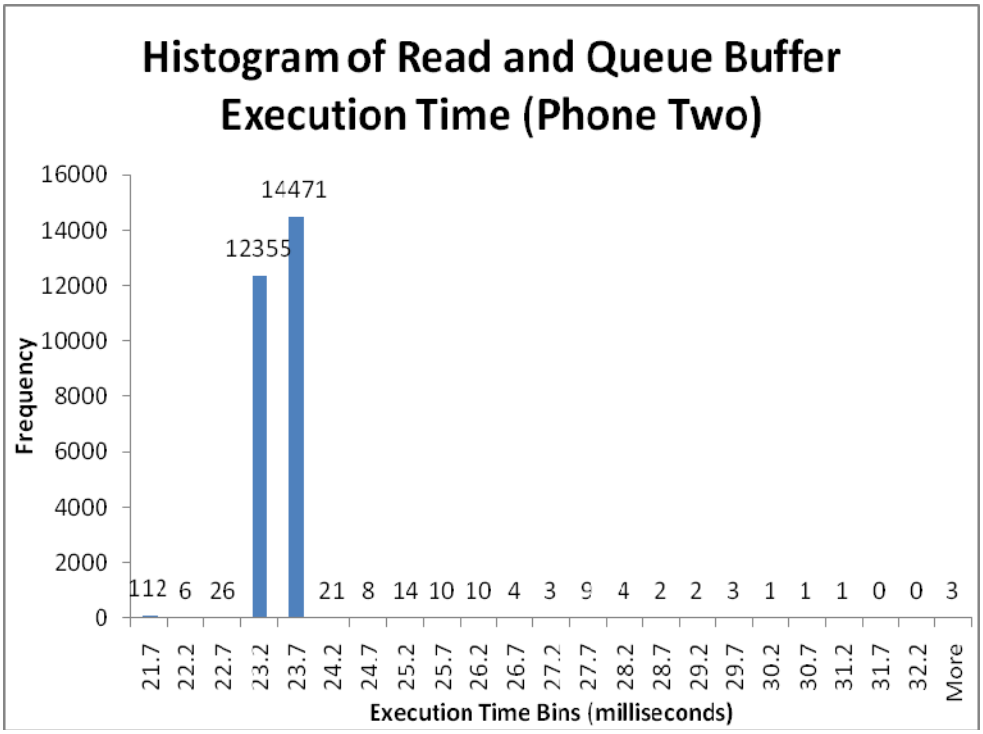

Figure 5.2 – Histogram of Time Difference Points Between Reading Audio Buffer on Phone Two.

The data can also be looked at from the perspective of differences in time compared to the time of the test as shown in Figure 5.3 for phone one and 5.4 for phone two.

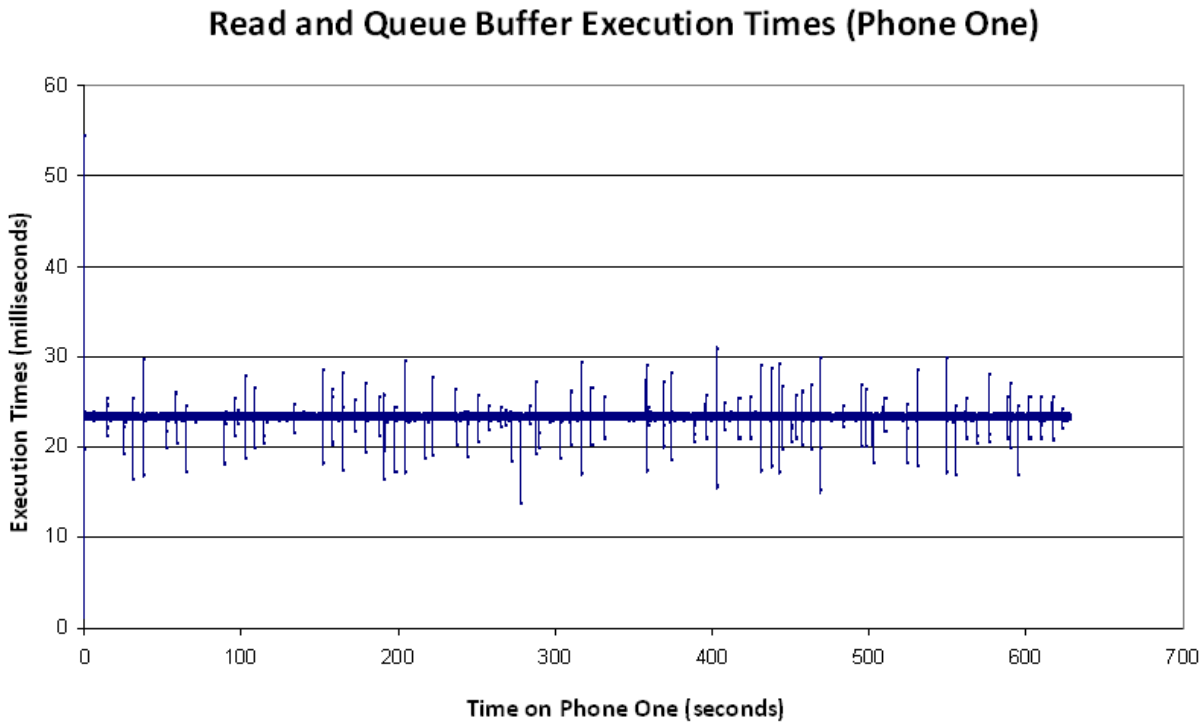

Figure 5.3 – Read and Queue Buffer Execution Time Differences for Phone One.

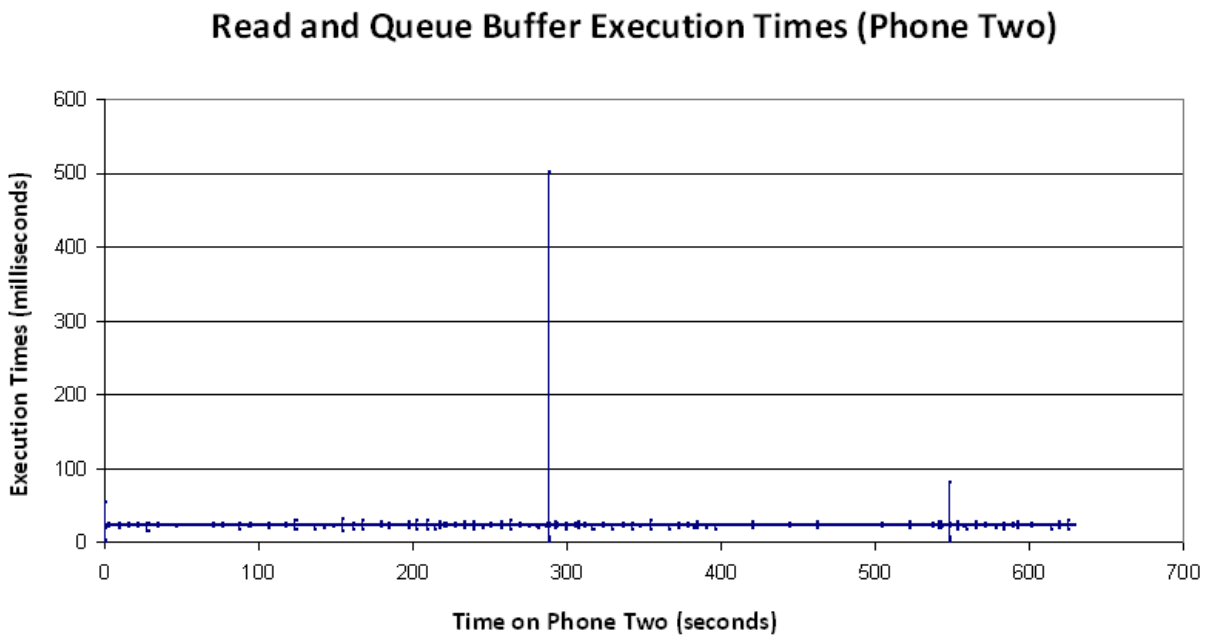

Figure 5.4 – Read and Queue Buffer Execution Time Differences for Phone Two.

Since the second phone had a major outlier (over 500 milliseconds) which prevents much of the graph from being seen, the same graph is presented again in Figure 5.5 with the outlier removed.

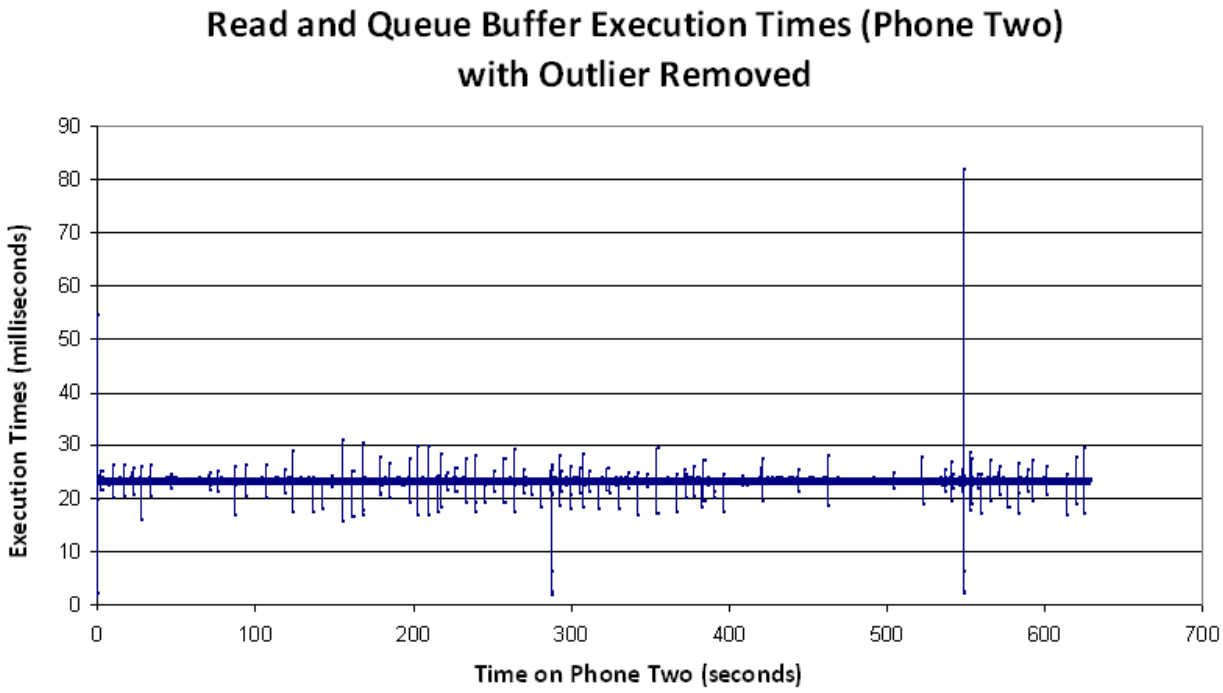

Figure 5.5 – Read and Queue Buffer Execution Time Differences for Phone Two with 500 Millisecond Outlier Removed.

The distribution of the data can also be seen through box plots as shown in Figure 5.6 for phone one and 5.7 for phone two.

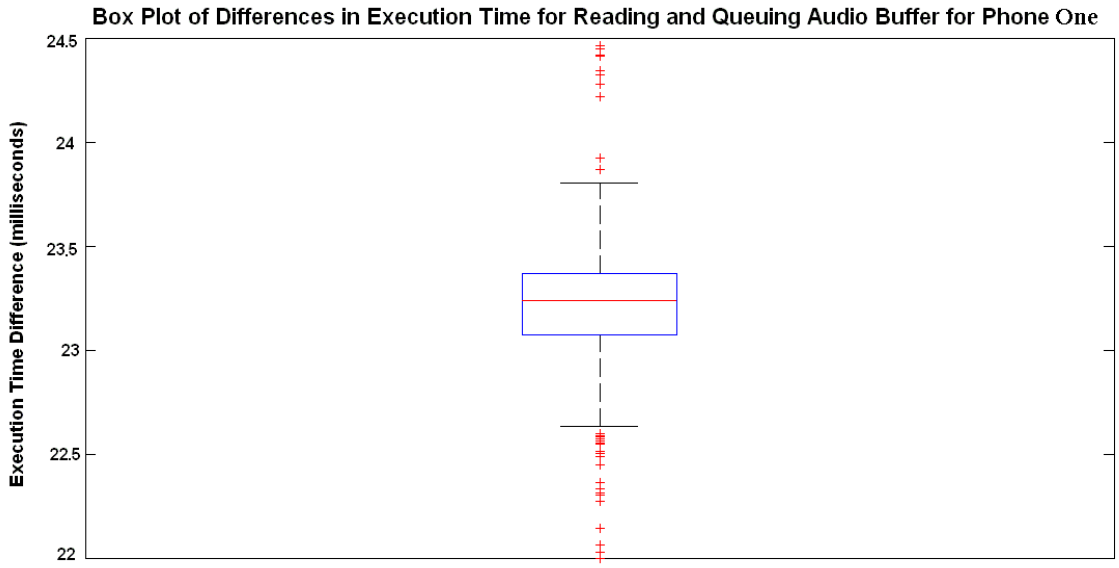

Figure 5.6– Box Plot of Differences in Execution Times for Reading and Queuing the Audio Buffer For Phone One.

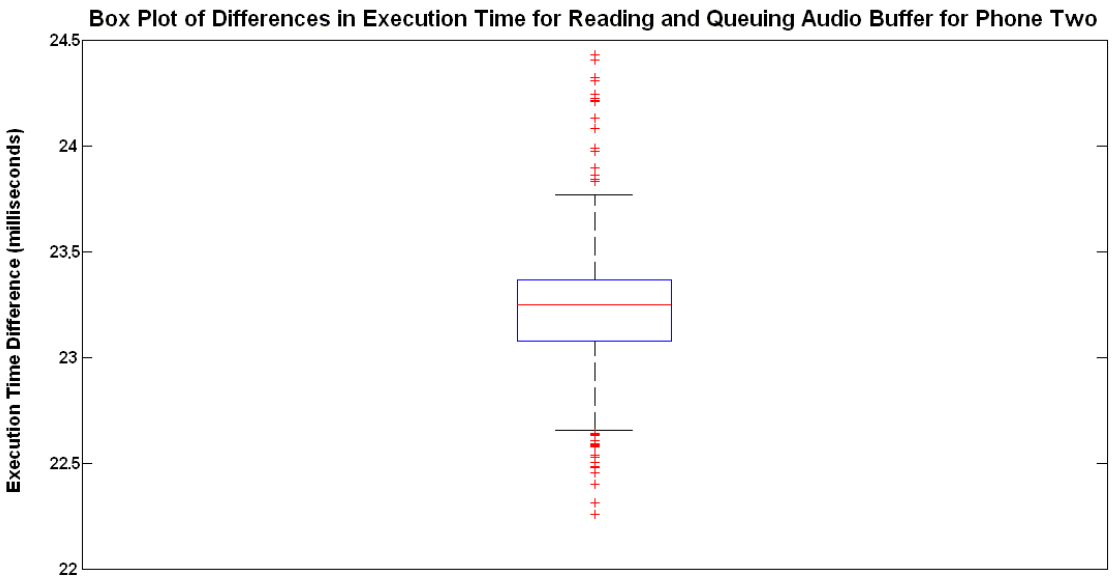

Figure 5.7– Box Plot of Differences in Execution Times for Reading and Queuing the Audio Buffer For Phone Two.

Some of the key values for each of the phones' time differences are shown in Tables 5.1 and 5.2.

Table 5.1 – Mean, Median, Minimum, Maximum, and Standard Deviation of the Time Differences in Execution of Reading and Queuing the Audio Buffer on Phone One (in milliseconds)

Mean	Median	Minimum	Maximum	Standard Deviation
23.215	23.238	1.976	54.316	0.419

Table 5.2– Mean, Median, Minimum, Maximum, and Standard Deviation of the Time Differences in Execution of Reading and Queuing the Audio Buffer on Phone Two (in milliseconds)

Mean	Median	Minimum	Maximum	Standard Deviation
23.216	23.247	1.832	502.967	3.037

It can also be useful to look at the relationship between successive read and queue buffer execution time intervals. For Figures 5.8 and 5.9, each point has an x-value representing a read and queue buffer execution time. The y-value for each point is the read and queue buffer execution time which immediately follows the x-value.

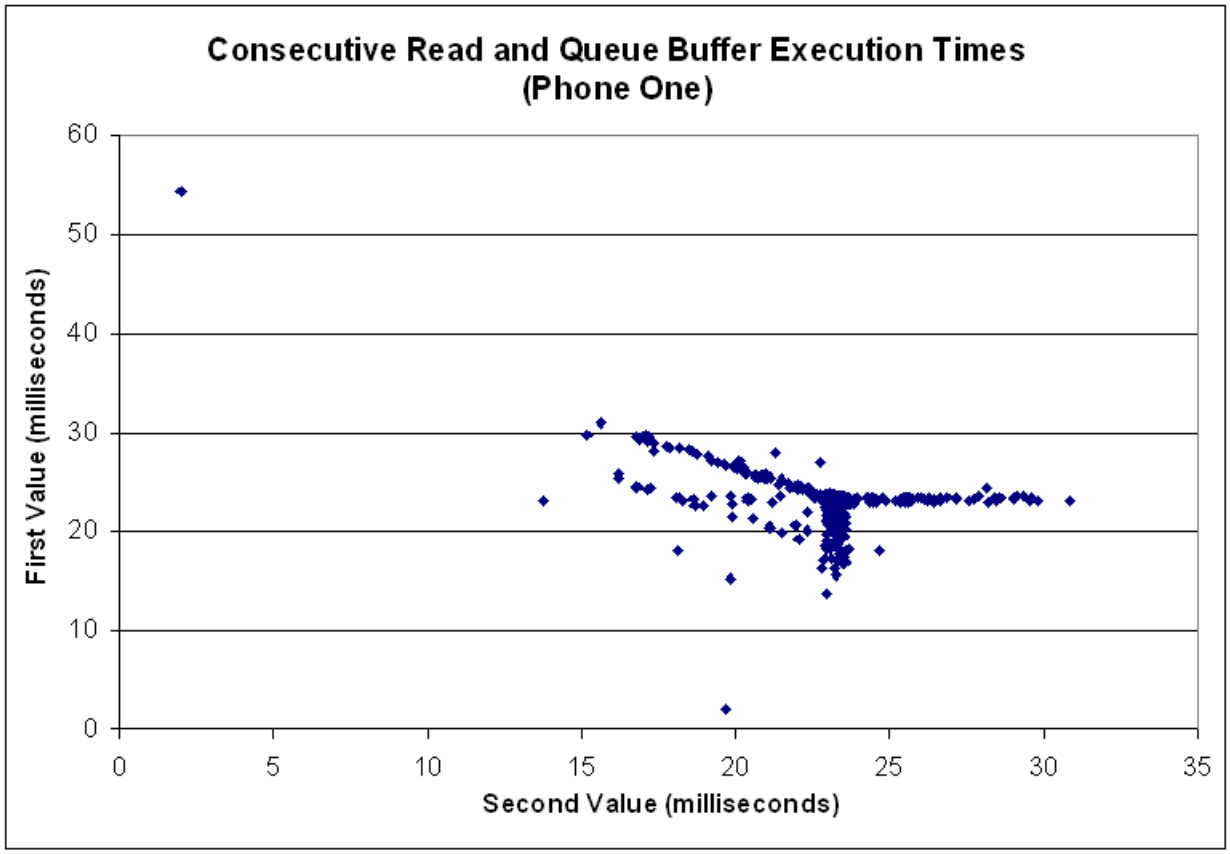

Figure 5.8 – Consecutive Read and Queue Buffer Execution Times from Phone One. The First Read and Queue Execution Times are Plotted on the Y-axis with Their Succeeding Value on the X-axis.

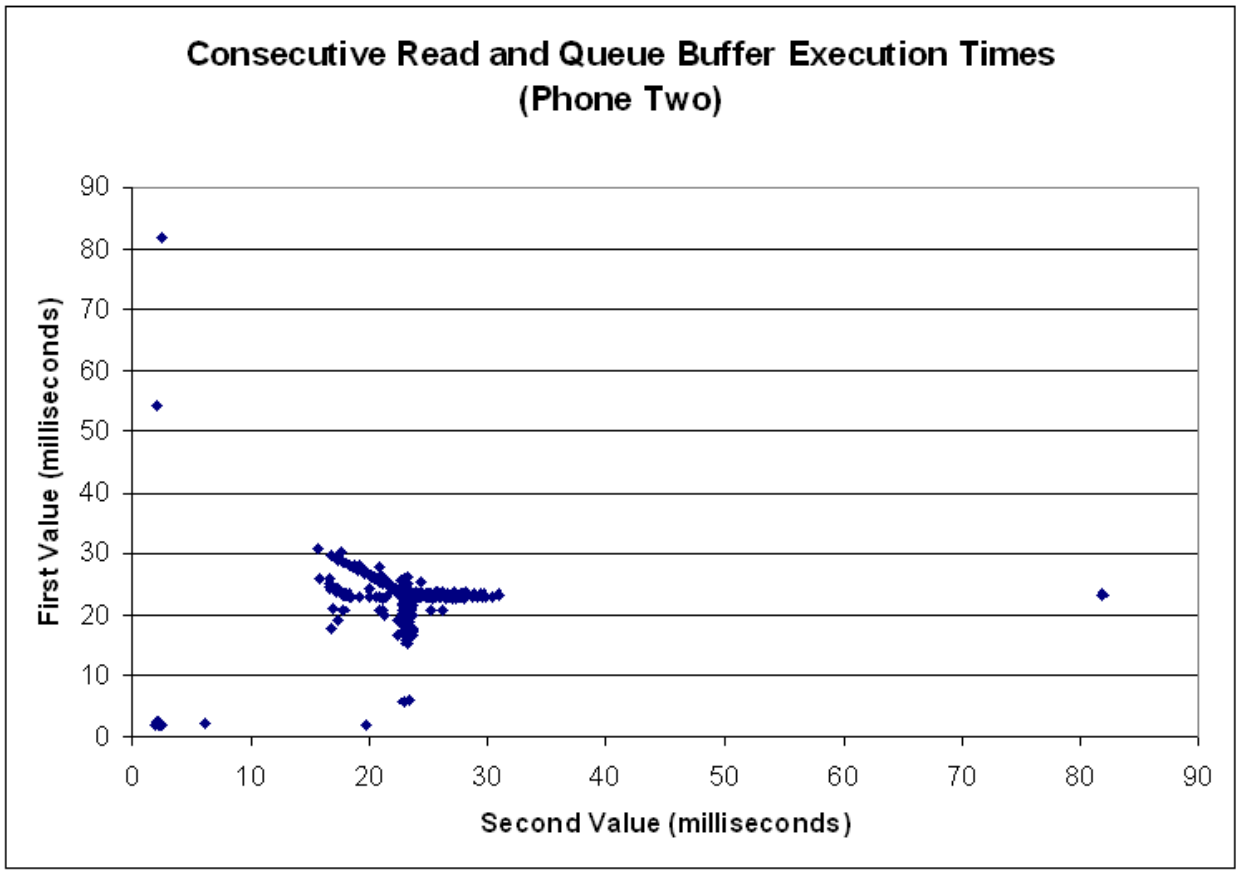

Figure 5.9 – Consecutive Read and Queue Buffer Execution Times from Phone Two. The First Read and Queue Execution Times are Plotted on the Y-axis with Their Succeeding Value on the X-axis.

5.3. Discussion

The ability to accurately detect location based on a time difference of arrival of a sound is heavily dependent on the precision of the time measurement (as described in the Background). One of the problems which may prevent the Openmokos from being able to accurately perform localization is a poor ability to read the time they were keeping. When reading the system clock after recording audio and comparing the time differences between system clock reads, there were infrequent but large variations in time differences even when running the program at the highest priority and changing the scheduling policy to use a first in first out method; one phone had

52.34 milliseconds difference between the highest and lowest values recorded and one phone showed over 500 milliseconds difference.

Perhaps what was most confusing about the results from the system time difference test were the shorter time differences. After the system time was read, 1024 samples of audio data were recorded at a sampling rate of 44,100 samples per second. This setup would imply that there would be at least 23.219 milliseconds of time difference between each system clock read, but both phones read some values that were lower than 2 milliseconds even though all the data were being recorded and sent (as indicated by the values received by MATLAB). Because the method for reading the system time is tied heavily to the system architecture and the documentation for the processor is under a non-disclosure agreement, it cannot be said for certain where these errors may come from, but one possible explanation for this discrepancy would be that the sound card records data into a ring buffer. When a request is sent to read data, some of the data may come from a time before the request was sent. This hypothesis is further supported by Figures 5.8 and 5.9, which show that, in general, a longer read and queue buffer execution time was generally followed by a shorter time, which may mean that the phone slowed down for one execution and had extra data available for the next execution. Shorter execution times, were generally followed by an execution time around 23.219 milliseconds, which may be an indication that the phones had caught up with the audio data that was available and then had to wait for new samples.

Although the outliers that occurred during the system time difference test are concerning and would cause problems when performing acoustic localization, there is no reason to indicate that with more investigation they could not be reduced. Over 99% of the time differences recorded were within .5 milliseconds of the expected value of 23.219 milliseconds.

6. Time Synchronization Using a Calibration Chirp

One method that could be used to synchronize the times between the phones is to use a single chirp to calibrate time between phones. There are two different approaches that were used to implement this technique. The first involved using an initial chirp to recalibrate the system clocks on the phones while the other method entailed correcting for differences in the clocks at the central processing point.

6.1. Recalibrating the System Clocks Based on a Chirp

In the first method of synchronization, a calibration chirp was played for both of the phones. The phones streamed audio and time data to MATLAB to be processed. After MATLAB determined when each phone heard the peak, it would calculate the difference between when each phone detected the peak. Then one phone was sent the difference in time. That phone would then set its time of day to correspond to the other phone's time of day.

6.1.1. Methods

The test took place in a confined room where the ambient noise was limited to 66 dB as measured by a Radio Shack Digital-Display Sound-Level Meter model 33-2055 (SPL meter). The physical set up consisted of placing each of the phones next to each other aligning them such that the ends of the phones with the microphones were touching and facing upwards. The sound source was a single speaker (Altec Lansing Computer Speaker System ACS90), and was aligned to face the phones and placed 1.05 meters away, perpendicular to the line that intersects the phones' microphones. The sound source was connected to a separate computer other than the central processing point, so that playing the chirp would not result in distortion brought on by the

heavy processing. The computer chosen for the sound source was a Dell desktop running Fedora 10 Linux with kernel version 2.26.27 with 2.8 gigahertz Pentium IV processor and 512 megabytes of RAM. The test scenario can be seen in Figure 6.1.

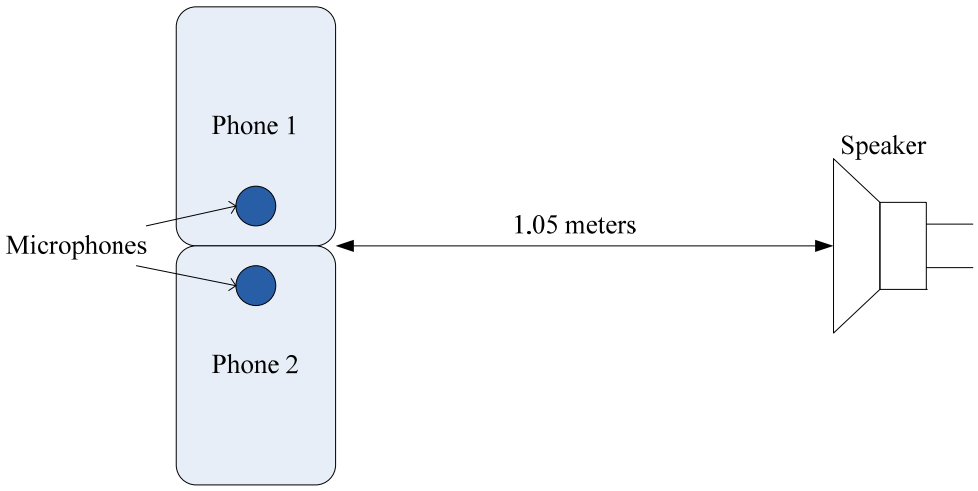

Figure 6.1 – Test Setup for Recalibrating the Phone System Clocks Based on a Calibration Chirp.

Based on this setup, errors could result from the alignment of the devices. By measuring the physical distances from each of the microphones to the sound source with a tape measure, the distances were known to be equal to within a centimeter. A one centimeter difference will produce less than thirty microseconds of time difference.

The sound source was set at a volume such that when a five kilohertz sine wave was played, the sound intensity at the edge of the phone cases was one hundred ten decibels sound pressure level. The C code on the phones was set to run for three MATLAB frames, approximately 19.29 seconds in which the phones listened for a one hundred millisecond chirp played by the sound source. The phones sent the time stamped audio data to the central processing point where the time difference of sound arrival of this chirp was computed by MATLAB and sent back to the phone that was behind in time. A value of zero was sent to the other phone. Both phones added their received time delay to their system clocks, thereby, creating an offset on one of the phones.

Following this procedure, additional data were collected to test the performance of the time synchronization between the phones.

6.1.2. Results

Figure 6.2 plots the time difference results from the chirp detections after the calibration chirp was used to synchronize times between the phones. The plot displays the time at which phone one's data recognized the chirp subtracted by the time at which phone two's data recognized the chirp with respect to the time of phone one.

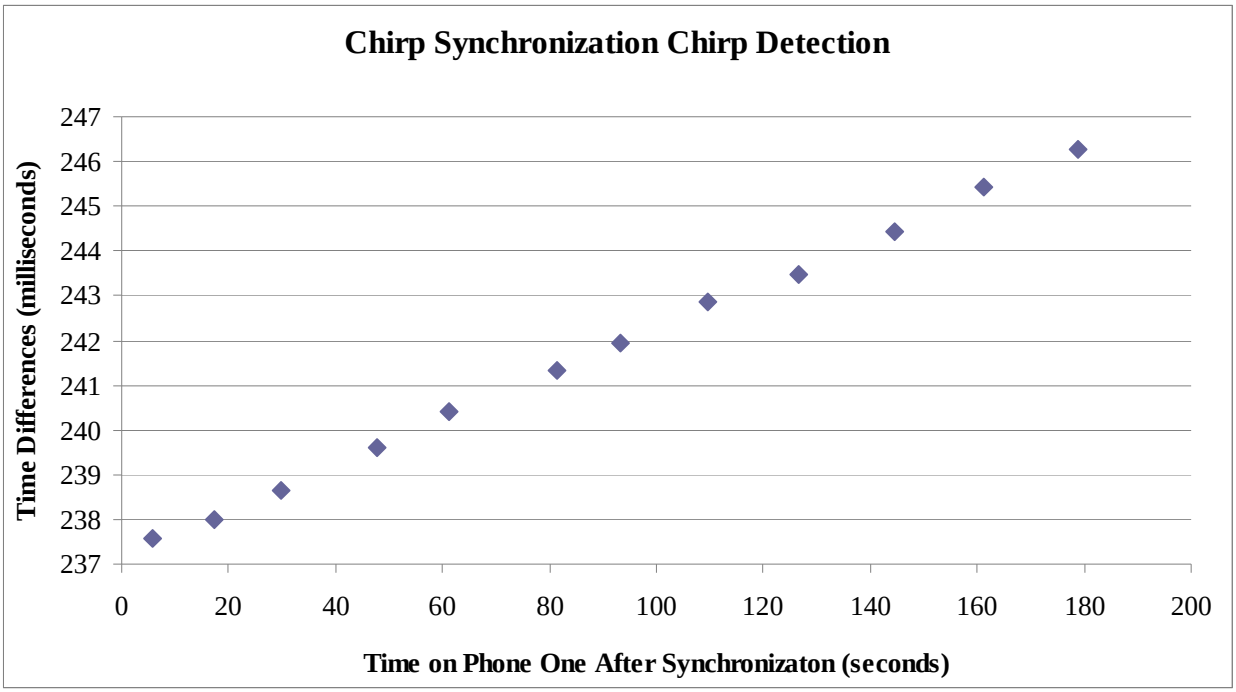

Figure 6.2 – Time Differences of Chirp Detections that Used a Calibration Chirp for Initial Time Synchronization.

The information in the previous plot indicated that the time difference was drifting over time. The drift rates were calculated by dividing the rates at which the time difference changed between two chirp detections by the time passed between those chirp occurrances. The mean of the drift rates was then calculated to be:

Average Drift Rate = 50.2891 Microseconds of Drift per Second

Using Figure 6.2 the y-intercept could be used to determine the time synchronization between the phones at time t = 0. This value indicates the time difference between the phones when they were first synchronized. Microsoft Excel 2007 best line fit feature was used to determine the y-intercept and it was calculated to be:

Time Difference at Zero Seconds = 237.2 Milliseconds

6.1.3. Discussion

As shown on the time difference plot in Figure 6.2 the maximum time difference between the two phones was 246 milliseconds. Even with the minimum error of 238 milliseconds, this method of calibration would create a minimum distance error of 40 meters.

The times between the phones with this method of synchronization can be observed to drift apart linearly. For every second, the average drift rate was found to be approximately 50.2 microseconds. This drift could be the clocks on the phones or their A/D converters operating at different rates. If this situation was the case then one phone would gather more audio data over the same period of time than the other phone. With 50.2 microseconds of drift per second, one phone would capture, on average, an extra 2.2 samples per second.

6.2. Correcting for Disparity in Central Processing Point

As shown by the results of the recalibration of system clocks with a calibration chirp, the clocks were not synchronized adequately enough for most acoustic localization applications. The cause of this result could have involved the phones not being able to accurately set their system clock. Therefore, the method was changed to resolve any disparities in phone times solely in MATLAB, eliminating the transmission of the data back to the phones.

6.2.1. Methods

The physical setup, equipment and the place used for this test were the same as the previous test for chirp detection. The phones listened for thirty chirps total and accordingly sent the time stamped audio data to the central processing point. The MATLAB code was changed, so that when the first calibration chirp was detected, it would calculate the time difference between the phones and add this time difference to one of the phone times for all subsequent detections. No change would be made to the times on the phones themselves.

6.2.2. Results

Figure 6.3 plots the time difference results from the chirp detections after making the corrections in time in MATLAB. The plot displays the time at which phone one's data recognized the chirp subtracted by the time at which phone two's data recognized the chirp with respect to the time of phone two after synchronization. Phone two was used as a reference time in this case because its time was not altered in MATLAB.

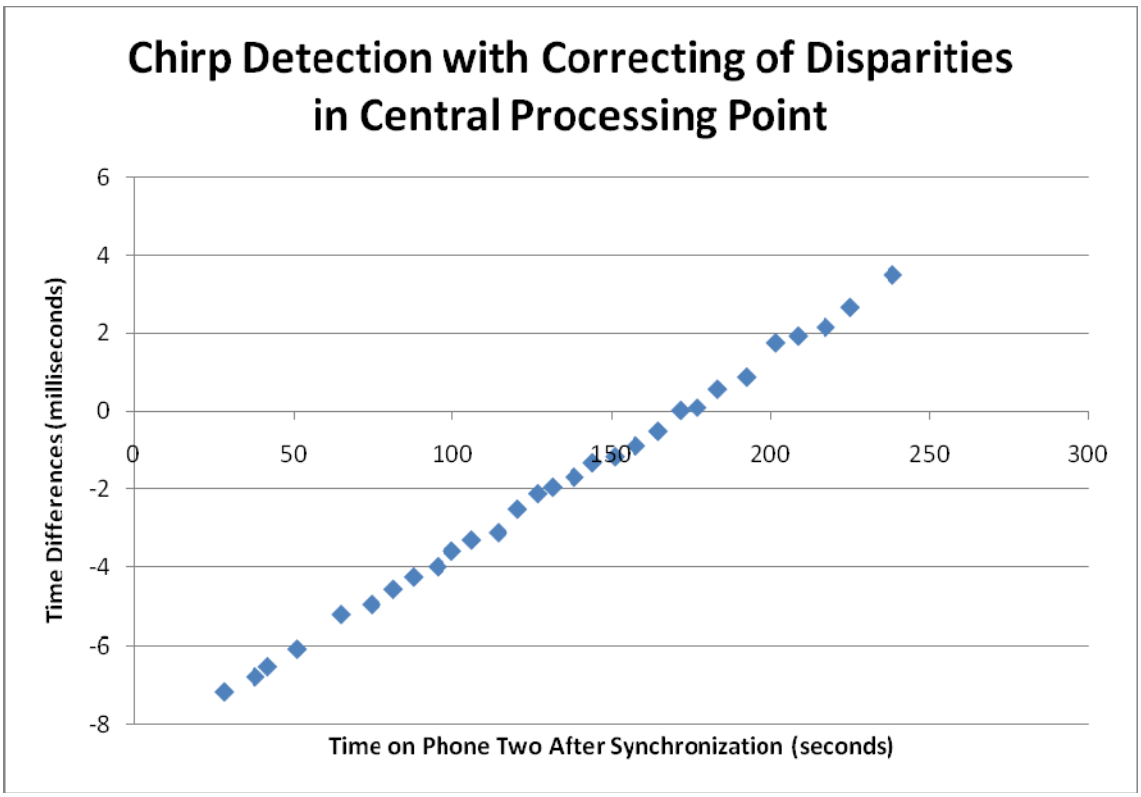

Figure 6.3 - Time Differences of Chirp Detections that Used Central Processing Point to Make Time Corrections.

The drift rate in this case was calculated to be:

Average Drift Rate = 50.4313 Microseconds of Drift per Second

Assuming a linear drift, the y-intercept (which is also the initial calibration precision) was:

Time Difference at Zero Seconds = -8.626 milliseconds

6.2.3. Discussion

The plot above shows the time differences between the phones with respect to the time on phone two after synchronization. After analyzing the data, it was discovered that at the first chirp detection the time from phone two was ahead of phone one. This result means that the current time difference was being added to phone one in MATLAB as it was behind in time. Therefore,

the time from phone two was the time used for synchronization and no change was being made to it.

The drift rate is certainly comparable between both of the tests using calibration chirp. For every second, the average drift rate of the recalibration using a calibration chirp test was found to be approximately the same as this test performance (50.2 microseconds per second for the previous test compared to 50.4 microseconds for this test). This result showed that the phones showed the same rate of error per second over time with different synchronization methods. The time synchronization between the phone's in this case was drastically improved. Although not enough tests were performed to be certain that setting the system clock performed worse than using an offset in MATLAB, it can at least be said that having the phone's attempting to correct their system clocks will not provide adequate synchronization for most applications.

7. Time Synchronization Using User Datagram Protocol Method

The last method for time synchronization involved using a User Datagram Protocol (UDP) Broadcast. The signal was broadcasted from the wireless router. The RF-waves emitted from the router transmitted the data at the speed of light (2.998×10^8 meters per second) through the air and to each of the phones. Signals broadcasted at such speeds could have produce errors on the order of nanoseconds (since the router was not aligned to be equidistant from the phones), but these errors were negligible to the tests. The main source of error was how quickly the phones' wireless cards could detect and interpret the signal. Each of the phones should have received the broadcast very close to the same time as one another, at which point the phones would reset their system clocks to a particular time.

7.1. Methods

The UDP broadcast method of time synchronization was implemented using two different C programs. One of the programs was written to send the UDP broadcast; the other was used to receive the broadcast. The program used to send the UDP broadcast was executed on the desktop PC that was used previously for the sound source. This computer was used because it was already available in the testing environment and could be easily connected to the network. The code that sent the broadcast sent a particular data value to the wireless router's broadcast IP address – allowing the wireless router to broadcast the data.

The code used to receive the broadcast, which was on each of the phones, was first executed before the broadcast script was executed. The receive code first set its scheduling priority to the highest possible value and a scheduling policy to be first in first out. It did so with the sched_setscheduler function. According to the Linux man page, this function creates a

preemptive scheduling policy where by the process will always preempt any other process that may attempt to run [*SCHED_SETSCHEDULER Man Page*, 2008]. This action did not allow other processes to interfere with its execution, eliminating the chance that another process could have offset the execution of the code and thus offset when the broadcast was received. The code was then set into a listening mode where it awaited the UDP broadcast. When the broadcast was received, the code then set the clock of the phone to a predefined value then exited.

The physical setup for determining the precision of the UDP time synchronization was identical to that of the chirp calibration method. The sound source was set at a volume such that when a five kilohertz sine wave was played, the sound intensity at the edge of the phone cases was 85 decibels as measured by the SPL meter. It was also recorded by the SPL meter that the ambient noise to the testing environment was 58 decibels. During the testing, a chirp that was one hundred milliseconds in length and varied linearly in frequency from one hundred to twenty thousand hertz was played. Both of the phones streamed their time stamped audio data to the central processing point. By calculating at what times the chirps occurred, the time synchronization of the phones was approximated. The chirp was played roughly every ten to fifteen seconds. This test was continued in intervals for approximately twenty minutes.

7.2. Results

Figure 7.1 plots the results from chirp detection tests, as described in the time synchronization section of the methods. Before the tests in the plot below were performed, a User Datagram Protocol (UDP) broadcast packet was sent to synchronize the times. The plot displays the time at which phone one's data recognized the chirp subtracted by the time at which phone two's data recognized the chirp with respect to the time of phone one.

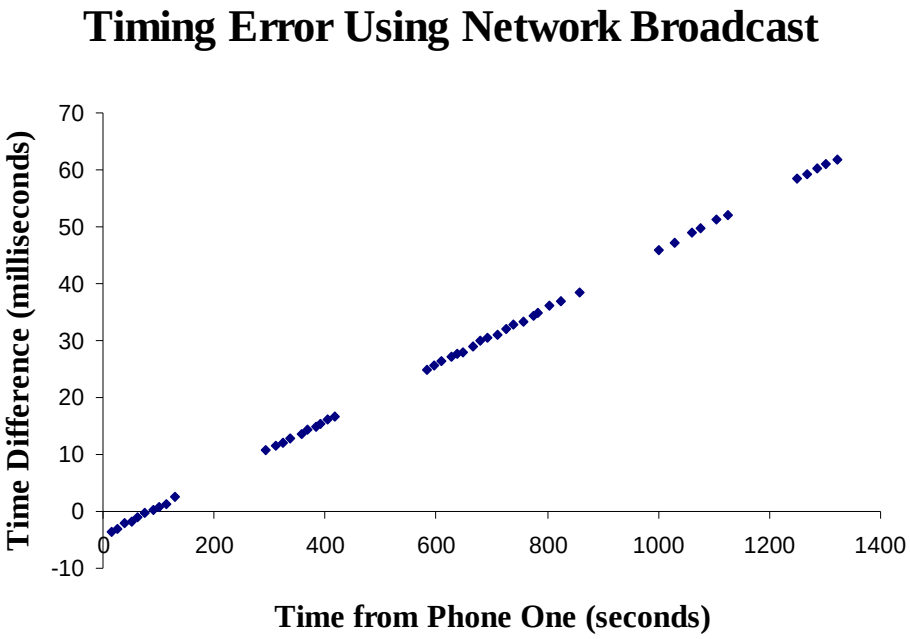

Figure 7.1 – Time Differences of Chirp Detections that Used a UDP Method for Initial Time Synchronization.

The information in Figure 7.1 indicated that the time difference was drifting over time. The drift rates were calculated by dividing the rates at which the time difference changed between two chirp detections by the time passed between the same two chirp occurrances. The mean of the drift rates was then calculated to be:

Average Drift Rate = 49.801 Microseconds of Drift per Second

Assuming a linear drift, the y-intercept (which is also the initial calibration precision) was:

Time Difference at Zero Seconds = -4.24 milliseconds

7.3. Discussion

The results from the UDP synchronization method turned out to be the most accurate of the methods used. Chirps were played during the test for approximately twenty minutes at varying intervals which allowed for analyzing different time difference measurements between the phones at which the sound arrived. As shown in the UDP time difference plot in Figure 7.1, this

method resulted in a maximum time difference of sixty one milliseconds over a 1308.156 second period. However, over the shorter period of time of less than two minutes this method produced results that were all less than 3.628 milliseconds apart. The time difference seems to grow linearly with time. This linearity could be exploited to improve the accuracy by predicting the relative times of the clocks. The results acquired through the UDP broadcast in general show that a single synchronization would be inadequate for the localization of a sound source; however, this method shows potential if the phones are resynchronized periodically.

7.4. UDP Method with Running Average

After the initial test of UDP time synchronization was shown to be the most accurate of the time synchronization methods, it was decided to use it with a moving average in attempt to eliminate the outliers shown in the read and queue execution time tests. The moving average would limit the affect the outliers had on their particular time stamp. This method of implementation was identical to the previous UDP synchronization test, with the exception of altered MATLAB code use on the central processing point. The MATLAB code kept track of the mean time difference. The code would then add the mean time difference to every new buffer to produce a new time stamp for the incoming audio data. Before performing the “chirping” portion of this test, three MATLAB buffers were run in attempt to prevent outliers in the first timestamps from affecting the test.

7.4.1. Results

Figure 7.2 plots the results from chirp detection tests. Before the tests in the plot below were performed, a User Datagram Protocol (UDP) broadcast packet was sent to synchronize the times.

The plot displays the time at which phone one's data recognized the chirp subtracted by the time at which phone two's data recognized the chirp with respect to the time of phone one.

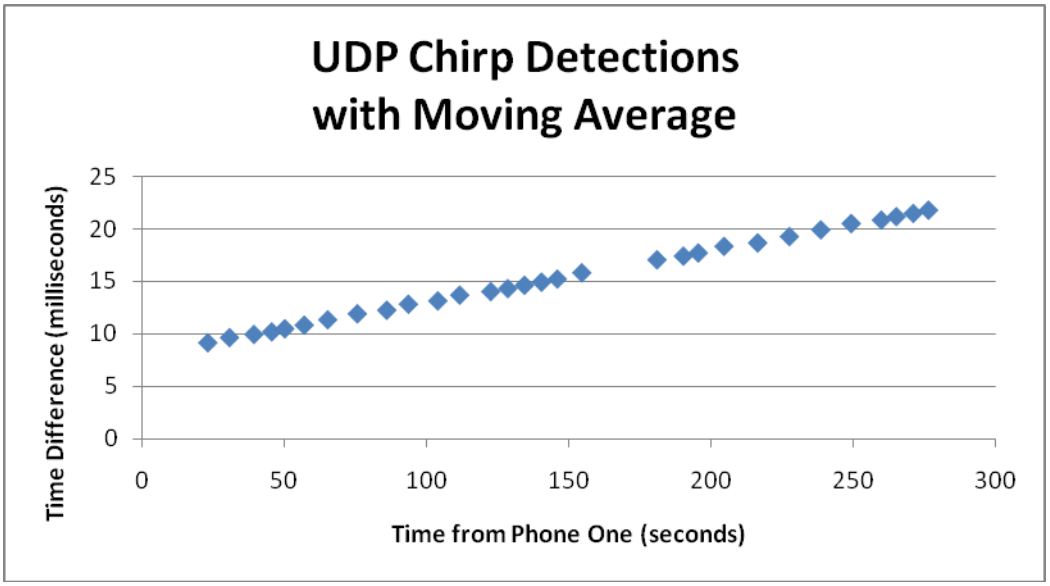

Figure 7.2 – Time Differences of Chirp Detections that Used a UDP Method for Initial Time Synchronization and Implemeted a Moving Average.

Figure 7.3 displays the rate of time difference change over time. The critical values of the data can be seen in Table 7.1.

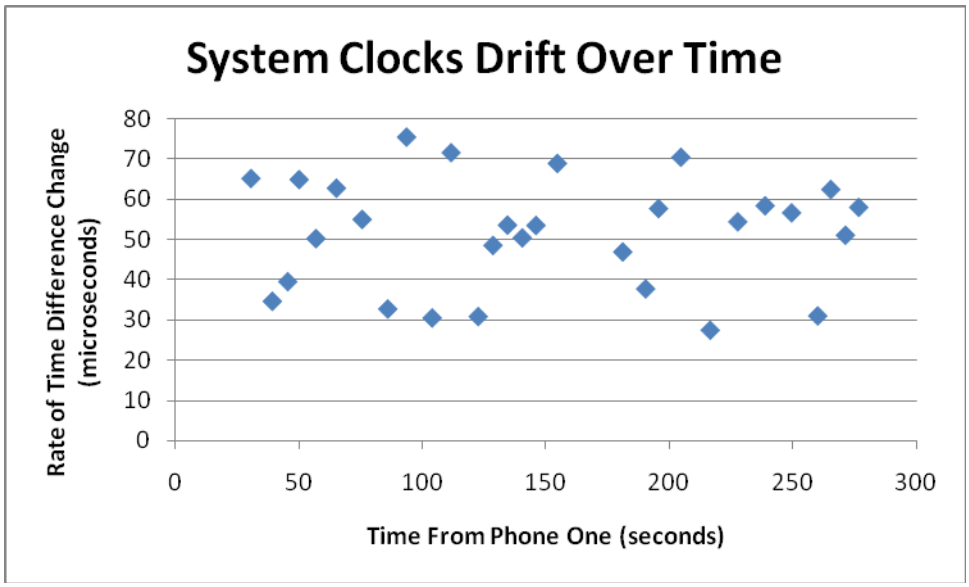

Figure 7.3 – Rate of Time Difference Drift of Chirp Detections that Used a UDP Method for Initial Time Synchronizationand Implemented a Moving Average

Table 7.1 – Critical Values of the Rate of Time Difference Change over Time for Chirp Detections

Maximum (microseconds)	Minimum(microseconds)	Mean(microseconds)
75.54763812	27.452213	51.790102

Assuming a linear drift, the y-intercept (which is also the initial calibration precision) was:

Time Difference at Zero Seconds = 7.98 milliseconds

7.4.2. Discussion

The results from the UDP synchronization with a moving average method did not yield all of the expected results. A period of twenty-three seconds was waited until chirps were played, which is reflected in Figure 7.2. The initial synchronization value was not as accurate as the previous UDP synchronization test, however not enough tests were performed to see whether the moving average was the cause of the discrepancy or whether synchronizing using a UDP broadcast packet can vary with the initial synchronization.

Another aspect that this test hoped to achieve was reducing the effect that read and queue outliers would have. However, no outliers occurred during the testing. The lack of outliers was to be expected, considering they occur very infrequently and only a portion of the C buffers time stamps are used as MATLAB buffer time stamps. This test leaves the question of a moving average implementation's ability to reduce or eliminate the effect of these outliers unanswered. In order to accurately test whether the moving average has the desired effect, further tests are needed.

7.5. Time Stamping Data after Reading Audio

After the previous tests, UDP time synchronization was still the most precise of the time synchronization methods. Therefore UDP synchronization was used as the basis for the next test. This method of implementation was identical to the previous UDP synchronization test, with the

exception of altering how both the C code on the phones and the MATLAB code used on the central processing point sent and received time stamps. In all of the previous implementations the phones read the time before the audio data were collected. In this implementation of UDP time synchronization, the C code was rewritten so that the timestamp could come after the corresponding audio data. This was an attempt to have the time stamp of the data better correspond to the audio data. In the read and queue buffer execution tests, the amount of time taken to capture a specified number of samples was sometimes less than the time needed to record the samples. One hypothesis for this discrepancy was that the audio buffer implemented by the ALSA drivers may retrieve samples on the phone from a ring buffer. If this was the case then reading the system clock for a time stamp before filling the audio buffers would create the potential for the audio buffer to use samples from before the time stamp was read, thus rendering the time stamp invalid. By having the time stamp after ALSA returns a full buffer of audio data, the last values of the audio data and the time stamp would correspond very well (as opposed to having no samples correspond to the timestamp).

7.5.1. Results

Figure 7.4 plots the results from chirp detection tests. Before the tests in the plot below were performed, a User Datagram Protocol (UDP) broadcast packet was sent to synchronize the times. The plot displays the time at which phone one's data recognized the chirp subtracted by the time at which phone two's data recognized the chirp with respect to the time of phone one.

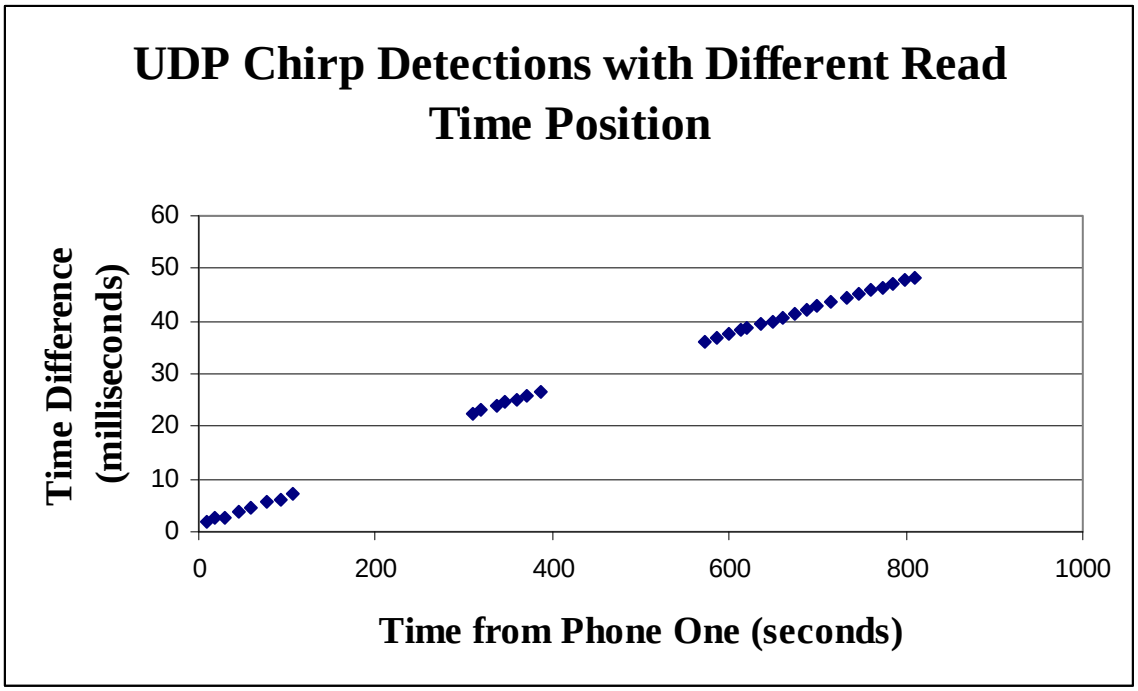

Figure 7.4 – Time Differences of Chirp Detections that Used a UDP Method for Initial Time Synchronization and Read Time at a Different Point in the Program.

The information from previous data indicated that the time difference was drifting over time. The drift rates for this test were calculated by dividing the rates at which the time difference changed between two chirp detections by the time passed between the same two chirp occurrances. The mean of the drift rates was then calculated to be:

Average Drift Rate = 53.31365856 Microseconds of Drift per Second

Assuming a linear drift, the y-intercept (which is also the initial calibration precision) was:

Time Difference at Zero Seconds = 2.27 Milliseconds

7.5.2. Discussion

The results from the UDP synchronization with obtaining the time stamp at a different point in the code were not conclusive. While the initial time difference is slightly better than before, not enough tests were performed to conclude that this was due to where the time was being read. Similarly to that of the moving average, no outliers occurred during this test. Because of the lack

of data, no statement can be made about whether reading the time after collecting audio data could benefit the time synchronization. Because of the low probability of observing an outlier, in order to tell whether this has the desired effect, further tests are needed that cannot be accomplished with the time constraints of this project.

8. General Discussion

During the development and testing of the acoustic localization system, the authors noted several points independent of specific tests which should be addressed. The issues of performing acoustic localization on other phone architectures, the benefits and detriments of writing code at a higher abstraction level, the stability of the wireless network, and the real-time performance of the system are addressed in the sections below.

8.1. Other Phone Architectures

Several drawbacks existed due to the choice of using the Openmoko as the phone for this project. One of the major drawbacks was the lack of floating point hardware. This limitation reduced the phones ability to implement an efficient method for cross-correlation. Any acoustic localization method which requires floating point math would either need to recast its algorithms to use fixed point or integer math, use a different phone, or stream all the data to a central point to be processed.

Another major drawback was the Openmoko's inability to take advantage of its GPS's PPS signal in order to keep accurate time. Unfortunately, it is unlikely that phones in the future will rectify this situation. The average consumer has no need for their phone to keep time accurate to within 100 nanoseconds, so dedicating a pin on the CPU for that purpose would be a questionable move by a phone manufacturer.

Finally, the ability of the phones to synchronize has not been completely proven. Tests have shown that the phones are able to initially synchronize to within ten milliseconds, but not enough has been done to show that this initial synchronization is repeatable.

8.2. Higher Level and Open Source Programming

The programming for the Openmoko was done in C using prebuilt libraries and drivers. This method significantly decreased the time it took to implement the program. On the other hand, by using these drivers and libraries, there was less control over how the hardware functioned. The libraries and drivers also led to lack of knowledge about how the hardware was interacting. In a real-time system, it is critical that the user have complete control over the hardware, but this is difficult to achieve this level of control with a preexisting operating system. In this project, writing drivers for the phone's audio system may have helped improve the time synchronization.

An additional consideration for developing a system using a phone as a sensor is whether or not to use open source software. Open source provides the benefit of being able to change or learn how any of the software works, providing unlimited possibilities for programming and configuration. Unfortunately, learning how any piece of software works can take a great deal of time. One issue frequently encountered by the authors with the Openmoko was poor or no documentation. Also, with open source software, there is no guarantee that anything will work as expected. At one point during the project, the phones became unable to boot, requiring a full reinstallation of the operating system.

8.3. Wireless Network Operation and Stability

During testing, the Openmokos did not have a problem sending the audio data over the wireless network at the required speeds. On occasion, data would have to be stored in the queue for a short duration of time but the system was always able to allocate sufficient space for the queue in memory. However, the wireless system was not completely stable. The wireless connection would intermittently disconnect and fail to reconnect unless the Openmokos were rebooted. Because of the random nature of the problem, no data were collected about the issue,

but the researchers observed that time between these disconnects would vary from approximately twenty minutes to several hours, and they occurred without regard to the data being transferred at the time. In order to develop a reliable system for deployment, these wireless issues would have to be resolved.

One way to reduce the phone's need to use the wireless would be to do all the sound detection on the phone itself. As mentioned before, the Openmokos lack floating point hardware. This situation makes it difficult to perform the cross-correlation since the efficient methods require the use of FFTs. Implementing a fixed point or integer FFT algorithm would greatly increase the complexity of the project. But it is not unreasonable to believe that a different phone architecture in the future would have the ability to perform the needed operations. Performing the calculations would trade the need for network bandwidth and stability with the need for processing power.

8.4. Real-Time Operation

In order to understand how this acoustic localization system would work in real-time, a distinction needs to be made about which parts of the processing need to operate in hard real-time and which can operate in soft real-time. All of the networking, cross-correlation, peak detection, and acoustic localization can be performed in soft real-time. As long as one can process data at the rate that it comes in on average, the system will remain functional. Buffers are used to allow for situations where the system may get behind due to the operating system servicing other processes or network delays. For these soft real-time processes, the system described in this paper worked in real-time. The programs ran for hours with no substantial increase in buffer sizes.

The hard real-time parts of the system included the collection of sound data and collecting time stamps for those data. It is here that the system performance could be improved. As mentioned earlier in the discussion of the system clock time difference test, there was variance in expected time stamp values and collected timestamp values. This system may be improved by linking the time stamp values directly with audio values as they are recorded in the audio drivers. The sound collection method developed here can also be improved. The sound collection operated in a sequential manner on blocks of ADC data. Time was recorded, followed by the audio data; then the data were queued up to be sent to the central processing point. However, during the period when the time was being recorded and the data were being queued up and sent there was a small amount of audio data that may have been lost. A quick solution to prevent some potential data loss would be to expand the buffer into which audio data are placed so that more can be recorded at once. A better solution, for future research, would be to ensure that the audio data was being read in a circular buffer whereby audio data can be read continuously. In this situation collection could occur while the time is being queried, and previous data is queued to be sent.

9. Conclusions and Recommendations for Future Work

There were four major accomplishments with this project. The first accomplishment was developing a system which was able to stream both audio data and timestamps in real-time over a network to be processed by a computer. The second accomplishment was implementing a real-time system to process and graph received audio data. The third accomplishment was detection of a sound; the implemented system was able to detect where a chirp occurred to within a window of 68 microseconds in some cases. The final accomplishment was testing and characterizing the nature of the time synchronization of the phones. With simplistic methods for synchronizing the times, the researchers were able to achieve synchronizations within ten milliseconds for 2-3 minutes at a time.

There are several possible ways to continue upon the research presented here. Earlier in this paper it was noted that the system presented in this paper may have breaks in the collection of the audio data. The current audio drivers used do not clearly define how one might create a circular buffer whereby audio data could be collected continuously (or if one is already being used). Although tests indicated that the amount of data that would be lost was generally small (around 22 samples for ever 23.22 milliseconds worth of data), the system should not lose data to ensure that it always is able to distinguish the sounds it is meant to detect. Furthermore, the current system needs to have its audio collection program operating at the highest priority, which may interfere with processes involved with time synchronization.

The timing synchronization methods developed for this system would be inadequate by themselves for synchronizing the phones over long periods of time. The 50 microseconds of drift for every second can very quickly prevent the system from accurately determining the location of

a sound source. If this system were to be developed for deployment, a method of synchronization would need to be employed that reduced drift and/or periodically resynchronized the phones.

Appendix A: MATLAB Code to Receive, Process, Plot, and Record Data on the PC

```
function varargout = time_test(varargin)
% TIME_TEST M-file for time_test.fig
%      TIME_TEST, by itself, creates a new TIME_TEST or raises
the existing
%      singleton*.
%
%      H = TIME_TEST returns the handle to a new TIME_TEST or
the handle to
%      the existing singleton*.
%
%      TIME_TEST('CALLBACK',hObject,eventData,handles,...) calls
the local
%      function named CALLBACK in TIME_TEST.M with the given
input arguments.
%
%      TIME_TEST('Property','Value',...) creates a new TIME_TEST
or raises the
%      existing singleton*.  Starting from the left, property
value pairs are
%      applied to the GUI before time_test_OpeningFcn gets
called.  An
%      unrecognized property name or invalid value makes
property
%      application
%      stop.  All inputs are passed to time_test_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI
allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help time_test

% Last Modified by GUIDE v2.5 21-Sep-2009 13:12:03

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @time_test_OpeningFcn, ...
                   'gui_OutputFcn',  @time_test_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
```

```
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State,
varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before time_test is made visible.
function time_test_OpeningFcn(hObject, eventdata, handles,
varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to time_test (see VARARGIN)

% Choose default command line output for time_test
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes time_test wait for user response (see UIRESUME)
% uiwait(handles.figure1);


% --- Outputs from this function are returned to the command
line.
function varargout = time_test_OutputFcn(hObject, eventdata,
handles)
% varargout  cell array for returning output args (see
VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

```
% --- Executes on button press in start.
function start_Callback(hObject, eventdata, handles)

warning off all

set(handles.start, 'UserData', 1);

T =0;
i=0;
d =0;

%Number of points to plot
plotpnt = 64;

%The wireless connection
port = get(handles.portnum, 'UserData');
t = tcpip('192.168.0.44', port);
set(t, 'InputBufferSize', 3000000);

t2 = tcpip('192.168.0.55', port);
set(t2, 'InputBufferSize', 3000000);

%Pre-compute chirp
time = .100;
ty=0:1/44100:time;
y = chirp(ty,100, time, 20000);
y = y';

M = length(y);
%N+M-1 needs to be a power of 2 for fast FFTs
N = 2^nextpow2(4*M)-M+1;

Y = fft(y, N+M-1);
YCONJ = conj(Y)';

%I wonder if anyone will ever actually read this code…
%First time through we have to zero pad...
first = zeros(1, M-1);
first2 = zeros(1, M-1);

%This is how many we will read at once
bufread = N*get(handles.nffts, 'UserData');

%Use about this much time
s = 2;
```

```
%seconds should be a multiple of the buffer, so that we can
scale
%appropriatly
a = s/(bufread/44100);
s = floor(a)*(bufread/44100);
%It does us no good if you can't see anything...
if(s == 0)
    s = (bufread/44100);
end
init_axes(handles, s, bufread, plotpnt)

%Open a file for writing
file = fopen('timedata.txt', 'a+');
file2 = fopen('timedata2.txt', 'a+');

counter = 1;
counter2 = 1;

stored = 0;
stored2 = 0;
run = 1;
run2 = 1;

%Let the Games Begin!
fopen(t)
fwrite(t, bufread, 'int32');

fopen(t2)
fwrite(t2, bufread, 'int32');

while(get(handles.start, 'UserData') == 1)

    if((get(t, 'BytesAvailable') >= (bufread*2+8)) &&  (get(t2,
'BytesAvailable') >= (bufread*2)+8))

        seconds = fread(t, 1, 'uint32');
        useconds = fread(t, 1, 'uint32');
        seconds2 = fread(t2, 1, 'uint32');
        useconds2 = fread(t2, 1, 'uint32');
        T = fread(t, bufread,'int16');
        T2 = fread(t2, bufread, 'int16');

        if(i >= s*44100/bufread)
           i = 0;
           init_axes(handles, s, bufread, plotpnt);
        end
```

```
        %%%%%%%%%%%%%%%%%%%%
        %%%%%%Phone 1%%%%%%%%
        %%%%%%%%%%%%%%%%%%%%

        %Buffer
        axes(handles.buff);
        plot(i*3+1, get(t, 'BytesAvailable'), '*')

%         %Signal
         axes(handles.signal);
         plot((1 + i*(length(T)/plotpnt):1 + length(T)/plotpnt +
i*length(T)/plotpnt), T(1:plotpnt:end))

        %Buffer Again
        axes(handles.buff);
        plot(i*3+2, get(t, 'BytesAvailable'), '*')

        %XCorr
        axes(handles.xcorr);
        d = fdxcorr(T, y, M, N, first, bufread);
        %Save values for the next run through
        first = T(length(T)-M+2:end)';
        [pltxcorr idexs] = sort(d, 'descend');
        resort = sort(idexs(1:(length(T)/plotpnt)+1));
        plot((1 + i*(length(T)/plotpnt):1 + length(T)/plotpnt +
i*length(T)/plotpnt), d(resort))
        %plot((1 + i*(length(T)/plotpnt):1 + length(T)/plotpnt +
i*length(T)/plotpnt), d(1:plotpnt:end), '*')

        [D IX]  = findpeaks(d, 'minpeakheight', 1*10^6,
'minpeakdistance', 8000);
        if(stored && (length(D) < 1))
            sec = savesec;
            usec = saveusec;
            usec = usec + floor(22.675736961*saveix);
            while(usec > 1000000)
                usec = usec - 1000000;
                sec = sec + 1;
            end
            signalfile = fopen(['signal_' num2str(counter)
'.txt'], 'w+');
            fwrite(signalfile, saveT, 'int16');
            fclose(signalfile);
            peakfile = fopen(['chirp_' num2str(counter)
'.txt'],'w+');
            fwrite(peakfile, savesec, 'int32');
```

```
            fwrite(peakfile, saveusec, 'int32');
            fwrite(peakfile, saved, 'double');
            fclose(peakfile);
            counter = counter + 1;
            fprintf(file,'%u, %u\n', sec, usec);
            stored = 0;
        end
        for j=1:length(D)
            if(stored && (IX(j) < 15000))
                if(saveval > D(j))
                    sec = savesec;
                    usec = saveusec;
                    usec = usec + floor(22.675736961*saveix);
                    while(usec > 1000000)
                        usec = usec - 1000000;
                        sec = sec + 1;
                    end
                    peakfile = fopen(['chirp_' num2str(counter)
'.txt'],'w+');
                    signalfile = fopen(['signal_'
num2str(counter) '.txt'], 'w+');
                    fwrite(signalfile, saveT, 'int16');
                    fclose(signalfile);
                    fwrite(peakfile, savesec, 'int32');
                    fwrite(peakfile, saveusec, 'int32');
                    fwrite(peakfile, saved, 'double');
                    fclose(peakfile);
                    counter = counter + 1;
                    fprintf(file,'%u, %u\n', sec, usec);
                else
                    stored = 0;
                end
            end
            if(stored && (IX(j) > 15000))
                stored = 0;
            end
           if(IX(j) > length(d)-15000)
                stored = 1;
                savesec = seconds;
                saveusec = useconds;
                saved = d;
                saveval = D;
                saveix = IX(j);
                saveT = T;
                run = 0;
           end
           if(stored == 0)
```

```
                sec = seconds;
                usec = useconds;
                usec = usec + floor(22.675736961*IX(j));
                while(usec > 1000000)
                    usec = usec - 1000000;
                    sec = sec + 1;
                end
                peakfile = fopen(['chirp_' num2str(counter)
'.txt'],'w+');
                signalfile = fopen(['signal_' num2str(counter)
'.txt'], 'w+');
                fwrite(signalfile, T, 'int16');
                fclose(signalfile);
                fwrite(peakfile, seconds, 'int32');
                fwrite(peakfile, useconds, 'int32');
                fwrite(peakfile, d, 'double');
                fclose(peakfile);
                counter = counter + 1;
                fprintf(file,'%u, %u\n', sec, usec);
                %fread(t, get(t, 'BytesAvailable'),'int8');
            end
            if(run)
             stored = 0;
            else
             run = 1;
            end
        end

        %plot(i, max(d), '*');

        %Buffer One Last Time
        axes(handles.buff);
        plot(i*3+3, get(t, 'BytesAvailable'), '*')


%          %%%%%%%%%%%%%%%%%%%%
%          %%%%%Phone 2%%%%%%%%
           %%%%%%%%%%%%%%%%%%%%

        %Buffer
        axes(handles.buff2);
        plot(i*3+1, get(t2, 'BytesAvailable'), '*')

%         %Signal
         axes(handles.signal2);
         plot((1 + i*(length(T2)/plotpnt):1 + length(T2)/plotpnt
+ i*length(T2)/plotpnt), T2(1:plotpnt:end))
```

```
%         %Buffer Again
%         axes(handles.buff2);
%         plot(i*3+2, get(t2, 'BytesAvailable'), '*')

%         %XCorr
          axes(handles.xcorr2);
          d = fdxcorr(T2, y, M, N, first2, bufread);
          %Save values for the next run through
          first = T2(length(T2)-M+2:end)';
          [pltxcorr idexs] = sort(d, 'descend');
          resort = sort(idexs(1:(length(T)/plotpnt + 1)));
          plot((1 + i*(length(T)/plotpnt):1 + length(T)/plotpnt
+ i*length(T)/plotpnt), d(resort))
          %plot((1 + i*(length(T2)/plotpnt):1 +
length(T2)/plotpnt + i*length(T2)/plotpnt), d(1:plotpnt:end),
'*')

          [D IX]  = findpeaks(d, 'minpeakheight', 1*10^6,
'minpeakdistance', 8000);
          if(stored2 && (length(D) < 1))
            sec = savesec2;
            usec = saveusec2;
            usec = usec + floor(22.675736961*saveix2);
            while(usec > 1000000)
                usec = usec - 1000000;
                sec = sec + 1;
            end
            peakfile = fopen(['chirp2_' num2str(counter2)
'.txt'],'w+');
            signalfile = fopen(['signal2_' num2str(counter2)
'.txt'], 'w+');
            fwrite(signalfile,saveT2, 'int16');
            fclose(signalfile);
            fwrite(peakfile, savesec2, 'int32');
            fwrite(peakfile, saveusec2, 'int32');
            fwrite(peakfile, saved2, 'double');
            fclose(peakfile);
            counter2 = counter2 + 1;
            fprintf(file2,'%u, %u\n', sec, usec);
            stored2 = 0;
          end
          for j=1:length(D)
              if(stored2 && (IX(j) < 15000))
                if(saveval2 > D(j))
                    sec = savesec2;
                    usec = saveusec2;
```

```
                    usec = usec + floor(22.675736961*saveix2);
                    while(usec > 1000000)
                        usec = usec - 1000000;
                        sec = sec + 1;
                    end
                    peakfile = fopen(['chirp2_'
num2str(counter2) '.txt'],'w+');
                    signalfile = fopen(['signal2_'
num2str(counter2) '.txt'], 'w+');
                    fwrite(signalfile, saveT2, 'int16');
                    fclose(signalfile);
                    fwrite(peakfile, savesec2, 'int32');
                    fwrite(peakfile, saveusec2, 'int32');
                    fwrite(peakfile, saved2, 'double');
                    fclose(peakfile);
                    counter2 = counter2 + 1;
                    fprintf(file2,'%u, %u\n', sec, usec);
                else
                    stored2 = 0;
                end
            end
            if(stored2 && (IX(j) > 15000))
                    stored2 = 0;
            end
        if(IX(j) > length(d)-15000)
            stored2 = 1;
            savesec2 = seconds2;
            saveusec2 = useconds2;
            saved2 = d;
            saveval2 = D;
            saveix2 = IX(j);
            saveT2 = T2;
            run2 = 0;
        end
        if(stored2 == 0)
            sec = seconds2;
            usec = useconds2;
            usec = usec + floor(22.675736961*IX(j));
            while(usec > 1000000)
              usec = usec - 1000000;
             sec = sec + 1;
            end
            peakfile = fopen(['chirp2_' num2str(counter2)
'.txt'],'w+');
            signalfile = fopen(['signal2_' num2str(counter2)
'.txt'], 'w+');
            fwrite(signalfile, T2, 'int16');
```

```
                fclose(signalfile);
                fwrite(peakfile, seconds2, 'int32');
                fwrite(peakfile, useconds2, 'int32');
                fwrite(peakfile, d, 'double');
                fclose(peakfile);
                counter2 = counter2 + 1;
                fprintf(file2,'%u, %u\n', sec, usec);
                %clear out the buffer, we have what we need,
lets not waste
                %memory
                %ead(t2, get(t, 'BytesAvailable'),'int8');

            end
            if(run2)
             stored2 = 0;
            else
             run2 = 1;
            end

        end

         %Buffer one last time
         axes(handles.buff2);
         plot(i*3+3, get(t2, 'BytesAvailable'), '*')

        drawnow

        i = i + 1;
    end
end

fclose(t);
fclose(t2);
fclose(file);
fclose(file2);


function [d] = fdxcorr(T, y, M, N, first, bufread)
    x = zeros(bufread/N, N+M-1);

    x(1,:) = [first T(1:N)'];

    for i = 2:(bufread/N)
       x(i,:) = T((i-1)*N-M+2:i*N)';
    end

    c = zeros(1, N+M-1);
```

```
    d = zeros(1, bufread);

    for i = 1:(bufread/N)
        %Matlab does some weird stuff with their
        %xcorr that should make it go faster
        p = xcorr(x(i,:), y);
        c = p(ceil(length(p)/2):end);
        d((i-1)*N+1:i*N) =  c(M:end);
    end

function init_axes(handles, s, bufread, plotpnt)

 upper = 44100*s/bufread;

 %Buffer info - Displays 3 times every run through
 axes(handles.buff);
 cla
 hold on
 axis([1 3*upper 0 3000000])

 %Signal - Displays every plotpnt-th (64th mostly) point every
run through
 axes(handles.signal);
 cla
 hold on
 axis([0 (bufread/plotpnt)*upper -10000 10000])


 %Cross-Correlation - one at a time for now
 axes(handles.xcorr);
 cla
 hold on
 axis([0 (bufread/plotpnt)*upper -2*10^6 1*10^7])

%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%Phone 2%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%
%Buffer
 axes(handles.buff2);
 cla
 hold on
 axis([1 3*upper 0 3000000])

 %Signal
 axes(handles.signal2);
 cla
 hold on
```

```
 axis([0 (bufread/plotpnt)*upper -10000 10000])

 %Xcorr
 axes(handles.xcorr2);
 cla
 hold on
 axis([0 (bufread/plotpnt)*upper -2*10^6 1*10^7])


% --- Executes on button press in stop.
function stop_Callback(hObject, eventdata, handles)

set(handles.start, 'UserData', 0)
guidata(hObject, handles);
% hObject    handle to stop (see GCBO)
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)



function portnum_Callback(hObject, eventdata, handles)
get(hObject, 'String')
port = str2double(get(hObject,'String'));
set(handles.portnum, 'UserData', port);
% hObject    handle to portnum (see GCBO)
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of portnum as
text
%        str2double(get(hObject,'String')) returns contents of
portnum as a double


% --- Executes during object creation, after setting all
properties.
function portnum_CreateFcn(hObject, eventdata, handles)
% hObject    handle to portnum (see GCBO)
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

% Hint: edit controls usually have a white background on
Windows.
```

```
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function nffts_Callback(hObject, eventdata, handles)
get(hObject, 'String')
nfft = str2double(get(hObject,'String'));
set(handles.nffts, 'UserData', nfft);
% hObject    handle to nffts (see GCBO)
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of nffts as text
%  str2double(get(hObject,'String')) returns contents of nffts
as a double


% --- Executes during object creation, after setting all
properties.
function nffts_CreateFcn(hObject, eventdata, handles)
% hObject    handle to nffts (see GCBO)
% eventdata  reserved - to be defined in a future version of
MATLAB
% handles    empty - handles not created until after all
CreateFcns called

% Hint: edit controls usually have a white background on
Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

Appendix B: C Code to Record and Send time and Audio Data from a Phone

```
/*

This example reads from the default PCM device

*/

/* Use the newer ALSA API */
#define ALSA_PCM_NEW_HW_PARAMS_API
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>
#include <sched.h>
#include <alsa/asoundlib.h>


typedef struct node{
  int size;
  char * data;
  struct node * next;
}sdata;

sdata * head = NULL;
sdata * tail = NULL;

void enqueue (sdata * addto) {
  addto->next = NULL;
  if(head == NULL) {
    head = addto;
    tail = addto;
  } else {
    tail->next = addto;
```

```
    tail = addto;
  }
}

int dequeue(int sock ) {

  if(head == NULL) {
    return 0;
  }

  sdata * savesdata = head;
  int err;
  err = send(sock,head->data, head->size, MSG_DONTWAIT);
  if(err < 0) {
    if(errno == EAGAIN) {
      printf("EAGAIN\n");
      return -1;
    } else {
      return -2;
    }
  } else {
    if(err == head->size) {
      head = head->next;
      free(savesdata->data);
      free(savesdata);
    } else { //Didn't send everything
      sdata * incompsnd = malloc(sizeof(sdata));
      if(incompsnd == NULL) return -3;
      incompsnd->next = head->next;
      incompsnd->data = malloc(head->size-err);
      if(incompsnd->data == NULL) return -3;
      incompsnd->size = head->size-err;
      memcpy(incompsnd->data,head->data+err,head->size-err);
      if(head == tail) tail = incompsnd;
      free(head->data);
      free(head);
      head = incompsnd;
    }
  }
  return 1;
```

```
}


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

  /*************Priority************************/
  struct sched_param priority;
  priority.sched_priority = 99;

  sched_setscheduler(0, SCHED_FIFO, &priority);


  /****************internets*******************/
     int sockfd, newsockfd, portno;
     socklen_t clilen;
     struct sockaddr_in serv_addr, cli_addr;

     if (argc < 2) {
         fprintf(stderr,"ERROR, no port provided\n");
         return 1;
     }
     sockfd = socket(AF_INET, SOCK_STREAM, 0);
     if (sockfd < 0) {
        printf("ERROR opening socket\n");
     return 1;
     }
     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     //printf("%d\n",portno);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     if (bind(sockfd, (struct sockaddr *)
&serv_addr,sizeof(serv_addr)) < 0) {
       printf("ERROR on binding\n");
       return 1;
     }
     listen(sockfd,5);
     clilen = sizeof(cli_addr);
     newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
&clilen);
```

```
    if (newsockfd < 0) {
      printf("ERROR on accept\n");
      return 1;
    }
    //printf("connection accepted\n");

  /****************sound***********************/

    int rc;
    int size;
    snd_pcm_t *handle;
    snd_pcm_hw_params_t *params;
    unsigned int val;
    int dir=0;
    snd_pcm_uframes_t frames;
    char *buffer;
    struct timeval tv;
    unsigned int seconds;
    unsigned int micro;
    unsigned int bufread = 0;
    unsigned int savbufread;
    int samp;

    int ret;

    //printf("starting sound stuff\n");

    /* Open PCM device for recording (capture). */
    rc = snd_pcm_open(&handle, "default",
SND_PCM_STREAM_CAPTURE, 0);
    if (rc < 0) {
      fprintf(stderr, "unable to open pcm device: %s\n",
snd_strerror(rc));
      exit(1);
    }

    /* Allocate a hardware parameters object. */
    snd_pcm_hw_params_alloca(&params);

    /* Fill it in with default values. */
    snd_pcm_hw_params_any(handle, params);
```

```
     /* Set the desired hardware parameters. */

     /* Interleaved mode */
     snd_pcm_hw_params_set_access(handle,
params,SND_PCM_ACCESS_RW_INTERLEAVED);

     /* Signed 16-bit little-endian format */
     snd_pcm_hw_params_set_format(handle,
params,SND_PCM_FORMAT_S16_BE);

     /* Two channels (stereo) */
     snd_pcm_hw_params_set_channels(handle, params, 1);

     /* 44100 bits/second sampling rate (CD quality) */
     val = 44100;
     snd_pcm_hw_params_set_rate_near(handle, params, &val,
&dir);

     /* Set period size to 32 frames. */
     frames = 32;
     snd_pcm_hw_params_set_period_size_near(handle, params,
&frames, &dir);

     /* Write the parameters to the driver */
     rc = snd_pcm_hw_params(handle, params);
     if (rc < 0) {
       fprintf(stderr, "unable to set hw parameters: %s\n",
snd_strerror(rc));
       exit(1);
     }

     /* Use a buffer large enough to hold one period */
     snd_pcm_hw_params_get_period_size(params, &frames, &dir);
     size = frames * 2; /* 2 bytes/sample, 1 channels */
     samp = size/2;

     /*********main part**********************************/

     snd_pcm_hw_params_get_period_time(params, &val, &dir);
```

```
rc = read(newsockfd, &savbufread, sizeof(int));
//endianness...
savbufread = (savbufread >> 24) |
  (( savbufread << 8) & 0x00FF0000) |
  ((savbufread >> 8 ) & 0x0000FF00) |
  (savbufread << 24);

if(rc < 0) {
  printf("Read error\n");
  return 1;
}

printf("Buffer Size: %d\n", savbufread);

while (1) {

  gettimeofday(&tv,NULL);
  seconds=tv.tv_sec;
  micro= tv.tv_usec;

  sdata * audiodata = malloc(sizeof(sdata));
  if(audiodata == NULL) {
 printf("Data Struct Not Allocated\n");
 return 1;
  }
  audiodata->data = malloc(size);
  if(audiodata->data == NULL) {
 printf("Data Not Allocated\n");
 return 1;
  }
  //This is our perfectly cromulent method for getting data
  rc = snd_pcm_readi(handle, audiodata->data, frames);

  if (rc == -EPIPE) {
 /* EPIPE means overrun */
 fprintf(stderr, "overrun occurred\n");
 snd_pcm_prepare(handle);
  } else if (rc < 0) {
 fprintf(stderr,"error from read: %s\n",snd_strerror(rc));
 printf("fail at 1");
  } else if (rc != (int)frames) {
```

```
      fprintf(stderr, "short read, read %d frames\n", rc);
      printf("fail at 2");
       }


       if(bufread < samp) {

      //First Part of Audio Data
      audiodata->size = 2*bufread;
      enqueue(audiodata);

      //Time Data
      sdata * timedata = malloc(sizeof(sdata));
      if(timedata == NULL) {
        printf("Time Struct Not Allocated\n");
        return 1;
      }
      timedata->size = 2*sizeof(int);
      timedata->data = malloc(2*sizeof(int));
      if(timedata->data == NULL) {
        printf("Time Data Not Allocated\n");
        return 1;
      }

      micro = micro + 23*bufread; //An approximation, .3242630
us off per samp
      while(micro > 1000000) {
        micro = micro - 1000000;
        seconds++;
      }
      seconds = (seconds >> 24) |
        (( seconds << 8) & 0x00FF0000) |
        ((seconds >> 8 ) & 0x0000FF00) |
        (seconds << 24);
      micro = (micro >> 24) |
        ((micro << 8) & 0x00FF0000) |
        ((micro >> 8 ) & 0x0000FF00) |
        (micro << 24);
      memcpy(timedata->data, &seconds, sizeof(unsigned int));
      memcpy(timedata->data + sizeof(unsigned int), &micro,
sizeof(unsigned int));
```

```
      enqueue(timedata);

      sdata * secondaudiodata = malloc(sizeof(sdata));
      if(secondaudiodata == NULL) {
        printf("Second Audio Struct Not Allocated\n");
        return 1;
      }
      secondaudiodata->size = size-2*bufread;
      secondaudiodata->data = malloc(size-2*bufread);
      if(secondaudiodata->data == NULL) {
        printf("Second Audio Data Not Allocated\n");
        return 1;
      }

      memcpy(secondaudiodata->data, audiodata->data+2*bufread,
size-2*bufread);

      enqueue(secondaudiodata);
      bufread = savbufread - frames + bufread;

       } else { //No time data this buffer
      audiodata->size = size;
      enqueue(audiodata);

      bufread = bufread - frames;
       }

       do {
      ret = dequeue(newsockfd);
      if(ret < -1) {
        printf("Catastrophic Error in Send or Malloc\n");
        return 1;
      }
       } while(ret > 0);

     } //while(1)

     snd_pcm_drain(handle);
     snd_pcm_close(handle);
     free(buffer);
```

```
     return 0;
}


/****************************************/
//bzero function

extern void *memset(void *, int, size_t);

void bzero (void *to, size_t count)
{
  memset (to, 0, count);
}
```

Appendix C: Code on a Phone to Synchronize Phones Using UDP Broadcast

```
/*RECEIVER CODE*/
#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <time.h>
#include <sched.h>

#define PORT 2080
#define DEST_ADDR "192.168.0.255" //broadcast addr

int main(int argc, char *argv[])
{
  /*************Priority************************/
  struct sched_param priority;
  priority.sched_priority = 99;

  sched_setscheduler(0, SCHED_FIFO, &priority);

        int sockfd;
        char buf[10];
        struct sockaddr_in sendaddr;
        struct sockaddr_in recvaddr;
        int numbytes;
        int addr_len;
        int broadcast=1;

     struct timeval tv;
     int seconds=0;
     int micro= 0;
     tv.tv_sec=seconds;
     tv.tv_usec=micro;
     time_t curtime;


        if((sockfd = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
        {
                perror("socket");
                exit(1);
```

```
        }
       if((setsockopt(sockfd,SOL_SOCKET,SO_BROADCAST,
                              &broadcast,sizeof broadcast)) ==
-1)
        {
                perror("setsockopt - SO_SOCKET ");
                exit(1);
        }

        printf("Socket created\n");

        sendaddr.sin_family = AF_INET;
        sendaddr.sin_port = htons(PORT);
        sendaddr.sin_addr.s_addr = INADDR_ANY;
        memset(sendaddr.sin_zero,'\0', sizeof
sendaddr.sin_zero);


        recvaddr.sin_family = AF_INET;
        recvaddr.sin_port = htons(PORT);
        recvaddr.sin_addr.s_addr = INADDR_ANY;
        memset(recvaddr.sin_zero,'\0',sizeof recvaddr.sin_zero);
        if(bind(sockfd, (struct sockaddr*) &recvaddr, sizeof
recvaddr) == -1)
        {
                perror("bind");
                exit(1);
        }


        addr_len = sizeof sendaddr;
        if ((numbytes = recvfrom(sockfd, buf, sizeof buf, 0,
              (struct sockaddr *)&sendaddr, (socklen_t
*)&addr_len)) == -1)
        {

        perror("recvfrom");
     exit(1);
     }

     // printf("%s\n",buf);

     //received broadcast / change time
     settimeofday(&tv,NULL);

     gettimeofday(&tv,NULL);
     seconds=tv.tv_sec;
```

```
     micro= tv.tv_usec;


     printf("Time Changed to:\n seconds:%d,
useconds:%d\n",seconds, micro);

     //strftime(buffer,30,"%T.",localtime(&curtime));
     //printf("%s%ld\n",buffer,tv.tv_usec);

        return 0;
}
```

Appendix D: Code on a PC to Send UDP Broadcast for Synchronization

```
/*SENDER CODE*/

#include <sys/types.h>
#include <sys/socket.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <time.h>

#define PORT 2080
// #define SRC_ADDR "172.16.1.120"
#define DEST_ADDR "192.168.0.255"

int main(int argc, char *argv[])
{
        int sockfd;
        int broadcast=1;
        struct sockaddr_in sendaddr;
        struct sockaddr_in recvaddr;
        int numbytes;;

        if((sockfd = socket(PF_INET,SOCK_DGRAM,0)) == -1)
        {
                perror("sockfd");
                exit(1);
        }

       if((setsockopt(sockfd,SOL_SOCKET,SO_BROADCAST,
                              &broadcast,sizeof broadcast)) ==
-1)
        {
                perror("setsockopt - SO_SOCKET ");
                exit(1);
        }

        sendaddr.sin_family = AF_INET;
        sendaddr.sin_port = htons(PORT);
        sendaddr.sin_addr.s_addr = INADDR_ANY;
        memset(sendaddr.sin_zero,'\0',sizeof sendaddr.sin_zero);
```

```
        if(bind(sockfd, (struct sockaddr*) &sendaddr, sizeof
sendaddr) == -1)
        {
                perror("bind");
                exit(1);
        }

        recvaddr.sin_family = AF_INET;
        recvaddr.sin_port = htons(PORT);
        recvaddr.sin_addr.s_addr = inet_addr(DEST_ADDR);
        memset(recvaddr.sin_zero,'\0',sizeof recvaddr.sin_zero);

     //tell to reset time
     sendto(sockfd, "time", 4 , 0,(struct sockaddr *)&recvaddr,
sizeof recvaddr);


        close(sockfd);

        return 0;
}
```

References

3G. (2009). Retrieved September 7, 2009, from http://www.wireless.att.com/learn/why/technology/3g-umts.jsp.

Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., & Cayirci, E. (2002). Wireless sensor networks: A survey. *Computer Networks, 38*(4), 393-422.

Altec lansing computer speaker system ACS90. Retrieved 9/25, 2009, from http://support.dell.com/support/edocs/acc/61408/Specs.htm.

Android: Official website. (2009). Retrieved September 4, 2009, from http://www.android.com/about/.

ATR0630, ATR0635 ANTARIS 4 GPS single chips. Retrieved September 4, 2009, from http://web.archive.org/web/20071010080226/www.u-blox.com/products/Data_Sheets/ATR0630_35_SglChip_Data_Sheet(GPS.G4-X-06009).pdf.

Birchfield, S., & Gillmor, D. *Acoustic localization by accumulated correlation.* Retrieved September 7, 2009, from http://www.ces.clemson.edu/~stb/research/acousticloc/.

Bretaña, G. (1987). *Admiralty manual of navigation* (pp. 605-609) The Stationery Office.

Casey, J. (1893). Chapter 7 - the hyperbola. *A treatise on the analytical geometry of the point, line, circle, and conic sections, containing an account of its most recent extensions, with numerous examples* (pp. 209) London: Longman's, Green & Co.

Clark, M. (2003). Transport services and protocols. *Data networks, IP and the internet: Protocols, design and operation* (pp. 282). England: John Wiley & Sons Ltd.

Dell inspiron image. Retrieved October 13, 2009, from http://images.bilsimser.com/dell_inspiron.jpg.

Elali, T. S. (2004). Discrete systems and digital signal processing with MATLAB. (pp. 88).

FWG114pv2 image. Retrieved October 13, 2009, from http://kbserver.netgear.com/images/fwg114pv2.gif.

Howard, D. M., & Angus, J. A. S. (2006). The velocity of sound in air. *Acoustics and psychoacoustics* (3rd ed., pp. 7).

IEEE 802.11: LAN/MAN wireless LANS. (2008). Retrieved September 7, 2009, from http://standards.ieee.org/getieee802/802.11.html.

Karl Holger, W. A. (2005). Protocols based on Receiver/Receiver synchronization. *Protocols and architectures for wireless sensor networks* (pp. 221). England: John Wiley & Sons Ltd.

Lasko, T. A., & Hauser, S. E. (2001). Approximate string matching algorithms for limited-vocabulary OCR output correction., *4307* 232.

Madisetti, V., & Williams, D. B. (1998). Overlap-add and overlap-save methods for fast convolution. *The digital signal processing handbook* (pp. 8.1). Boca Raton, Florida: CRC Press LLC.

May 2008 national occupational employment and wage estimates. Retrieved September 4, 2009, from http://www.bls.gov/oes/2008/may/oes_nat.htm#b17-0000.

N900 technical specifications. (2009). Retrieved September 4, 2009, from http://maemo.nokia.com/n900/specifications/.

Openmoko. (2009). Retrieved September 4, 2009, from http://www.openmoko.com/.

Openmoko freerunner image. (2008). Retrieved October 13, 2009, from http://www.gadgetarena.com/gadget-content/uploads/2008/07/openmoko-neo-freerunner.jpg.

OpenmokoWiki. (2009). Retrieved September 4, 2009, from http://wiki.openmoko.org/wiki/Main_Page.

Perkins, C. (2001). *Ad hoc networking*. Upper Saddle River, New Jersey: Pearson Education.

PowerVR SGX graphics IP core family. (2009). Retrieved September 4, 2009, from http://www.imgtec.com/PowerVR/sgx.asp.

Römer, Kay P, Blum, Philipp and Meier, Lennart. (2005). Chapter 7 - time synchronization and calibration in wireless sensor networks. *Handbook of sensor networks: Algorithms and architectures* (pp. 202). Hoboken, New Jersey: John Wiley & Sons Inc.

SCHED_SETSCHEDULER(2) linux programmer's manual (2008). Retrieved October 13, 2009, from http://www.kernel.org/doc/man-pages/online/pages/man2/sched_setscheduler.2.html.

ShotSpotter testimonials. Retrieved September 7, 2009, from http://www.shotspotter.com/results/testimonials.html.

Sprint extends 4G leadership by announcing additional U.S. markets for sprint 4G. (2009). Retrieved September 7, 2009, from http://newsreleases.sprint.com/phoenix.zhtml?c=127149&p=irol-newsArticle_newsroom&ID=1319758&highlight=.

Tenney, S., Mays, B., Hillis, D., Tran-Luu, D., Houser, J., & Reiff, C. (2004). Acoustic mortar localization system - results from OIF. *Ft.Belvoir Defense Technical Information Center.*

U.S. army recognizes top ten greatest inventions of 2004.(2005, September 1, 2005). *Defense AT & L, 34*, 85.

Webster, J. G. (1999). Introduction to time transfer. *The measurement, instrumentation, and sensors handbook* (pp. 185). Boca Raton, Florida: CRC Press LLC.

WM8753L. (2008). Retrieved September 14, 2009, 2009, from http://www.wolfsonmicro.com/uploads/documents/en/WM8753.pdf.

Xu, G. (2007). Introduction. *GPS: Theory, algorithms, and applications* (2nd ed., pp. 1). Germany: Springer.