Yashar Bahman

Dr. Phillip Nico

Computer Engineering - Senior Project

6/1/2012

Final Report

Wireless Ad Hoc Networks for Mobile Devices via High Frequency Sound Waves

Chapter 1: Introduction

Problem Statement:

In the past 10 years, mobile devices have evolved from bulky, malfunctioning car phones to sleek new "smart phones". Since they are now capable of running games and other useful applications, they have increasingly become more like computers than phones. They are capable of doing almost everything a PC can do, with exception of one leisurely activity, real time multi-player gaming. Sure, there are turn-based games, like "Words with Friends", which push data from phone to phone like email, but it is nowhere near the level real time gaming requires. This is where my project comes into play. Through the creation of a wireless ad-hoc network, local phones can push data back and forth fast enough to create games which involve the full attention of all players present, without needing to wait for a notification. Furthermore, users will not have to go through complex setup procedures or exchange pairing ID's, as all of this will be done automatically.

Scope:

To accomplish this the phone's speaker/microphone will be treated as a modulator/demodulator (or modem for short). The idea is that the phone will generate sine waves at various frequencies to output them on the speaker on one end and also record audio with the microphone to analyze and decode it into data on the other.

After Completion:

Once complete, these functionalities will all be rolled up into one easy to use library named Wave, which can be used to send data back and forth between devices without the user needing to deal with any complex handshaking overhead (like pairing or key exchanges). Multiple devices will connect to one another simultaneously. This is possible because when one device is broadcasting, any number of devices can be listening. However, two quarters of working on the program was only enough to fully grasp the magnitude of the problem and a fully efficient solution for real-time applications still remains unfinished. Instead, as a proof of concept, I did implement a program called WaveChat allows users to chat with one another utilizing the Wave library as the sole means of communication.

Figure 0: WaveChat Logo

Chapter 2: Background

Understanding Sound Waves:

The first thing to understand before approaching a solution to this problem is what sound waves actually are. According to Merriam Webster, sound waves are defined as:

> *Longitudinal pressure waves in any material medium regardless of whether they constitute audible sound <earthquake waves and ultrasonic waves are sometimes called sound waves>.*

Although that definition is helpful, it is still not entirely clear. Let's start by defining a wave by what it does. Waves move energy from one place to another. This is kind of like throwing a baseball to a friend: you need to first add energy to the ball to get it moving, once it reaches your friend's glove the remaining energy will cause an impact which disperses in heat. That energy originated from your arm, but ended up in your friend's glove. Waves also transfer energy, but they do so without transmitting matter. Instead, they either require a medium to permeate through (pressure waves), or require nothing at all and can even travel through a vacuum (electromagnetic waves). Electromagnetic waves are outside the scope of this paper; however, it's worth mentioning that their ability to transmit energy without transmitting matter is what categorizes them as a wave.

Pressure waves move energy by oscillating a medium. But what does that really mean? Consider Newton's Cradle (Figure 1). If you've seen one before, you know when the ball in to the left is released, it will come in contact with the other balls and transfer its energy through all the balls and eventually cause the right-most ball to swing out, leaving the balls in the center completely still. This is very similar to how sound waves travel, but instead of metal balls, they are air molecules. When an oscillation is induced on the air molecules in contact with a speaker,

they bump into other molecules adjacent to them and those molecules collide with those adjacent to them and so on until the molecules no longer have enough energy to pass on.

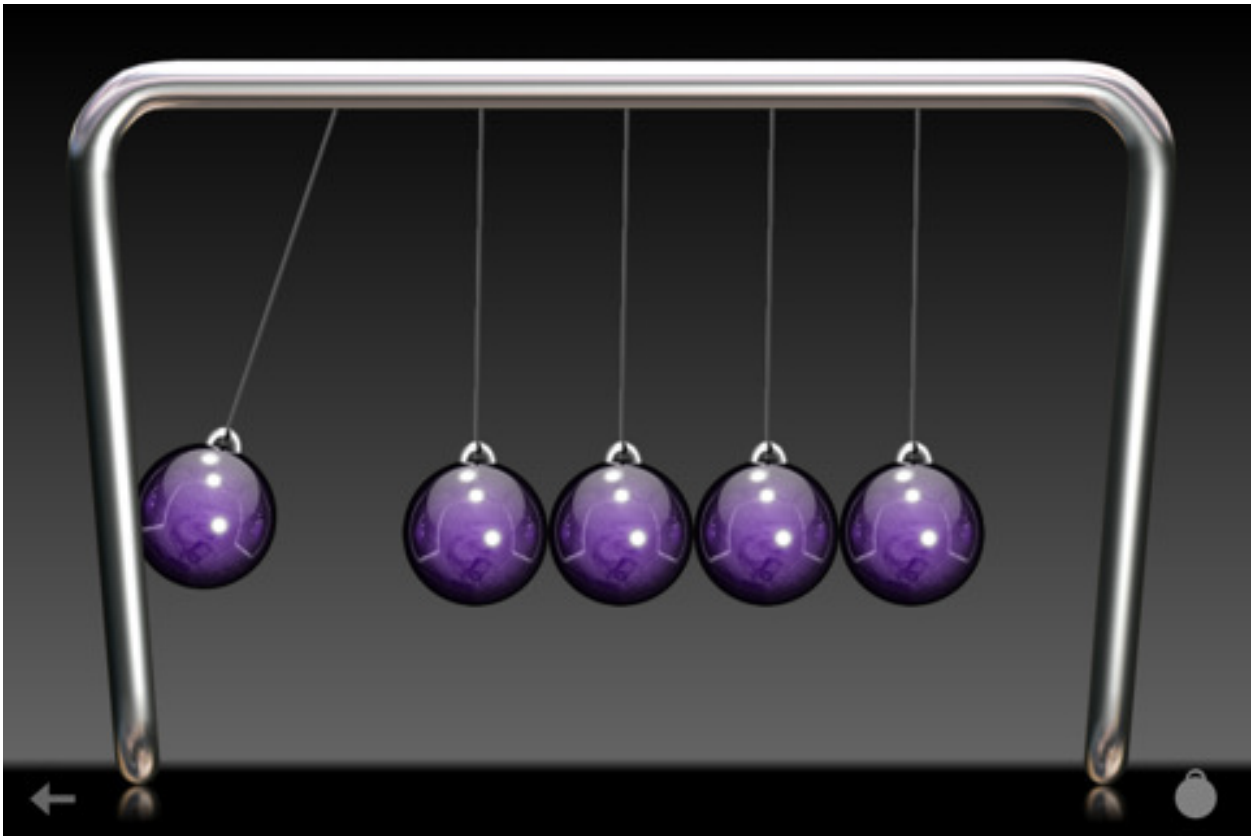

Figure 1: Newton's Cradle

If these molecular oscillations are within range, the human ear is able to interpret them as audible sound. But just like our eyes are not fast enough to see a hummingbird's wings flap, our ears also have their limits. The maximum range of the human ear is between 20 Hz to 22.05kHz; however, unless you're a musician or audio engineer, it is unlikely that you can hear anything above 18kHz.

Since mobile speakers are equipped with the ability to play tones above the average person's audible limitation, the extra bandwidth can be used to comfortably transmit data between devices. Since the tones are above 18kHz, the user will be completely unaware of the transfer.

Understanding Sound Generation:

In order to transmit data, it is necessary to create sound waves at certain frequencies. This is done with the phone's speaker. A speaker is essentially a cone that can move. Speakers can extend outward a certain length (the speaker to the left), retract inward (the speaker on the right), as well as stay anywhere in between. The position of the speaker is controlled by a speaker driver that is controlled programmatically. On Android devices, speakers generally have (2^{16}) discrete positions, with ($2^{15} - 1$) being the maximum extension, (0) being the middle, and (-2^{15}) being the maximum retraction. These drivers are designed to be able to output a maximum frequency of 22.05 kHz, which corresponds to the highest frequency that man can hear on average. To accomplish this, the drivers need to be able to write samples twice as fast, making 44,100 the maximum number of samples per second the driver can output.

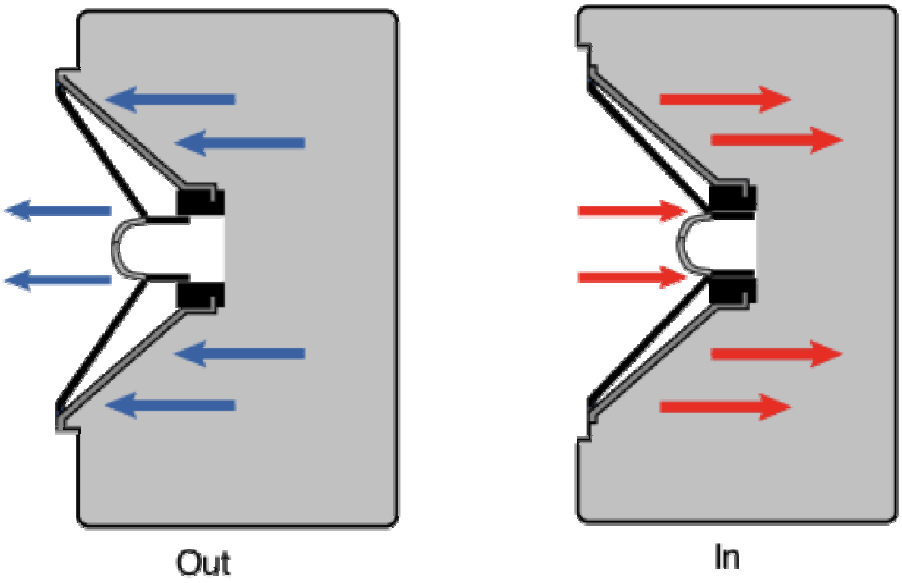

Figure 2: Speakers Out/In

Since the speaker's sample rate is limited, so is the waveform selection. Instead of using simple triangle or square waves, a sine wave must be used. This makes things more computationally intensive, but more on that later.

In order to generate a 1,000 Hz tone, the program needs to instruct the speaker to move into various positions at the right time. To do this with a sine wave, the equation will look like this:

$$y = \sin\left(2\pi x * \frac{frequency}{sampleRate}\right) * maxSpeakerExtention$$

Or more specifically:

$$y = \sin\left(2\pi x * \frac{1000}{44100}\right) * 2^{15}$$

Where y is the speaker position, x is the index, and both x and y must be whole numbers. This will result in a buffer like the one graphed in Figure 3 as well as a smoothly generated sine wave.

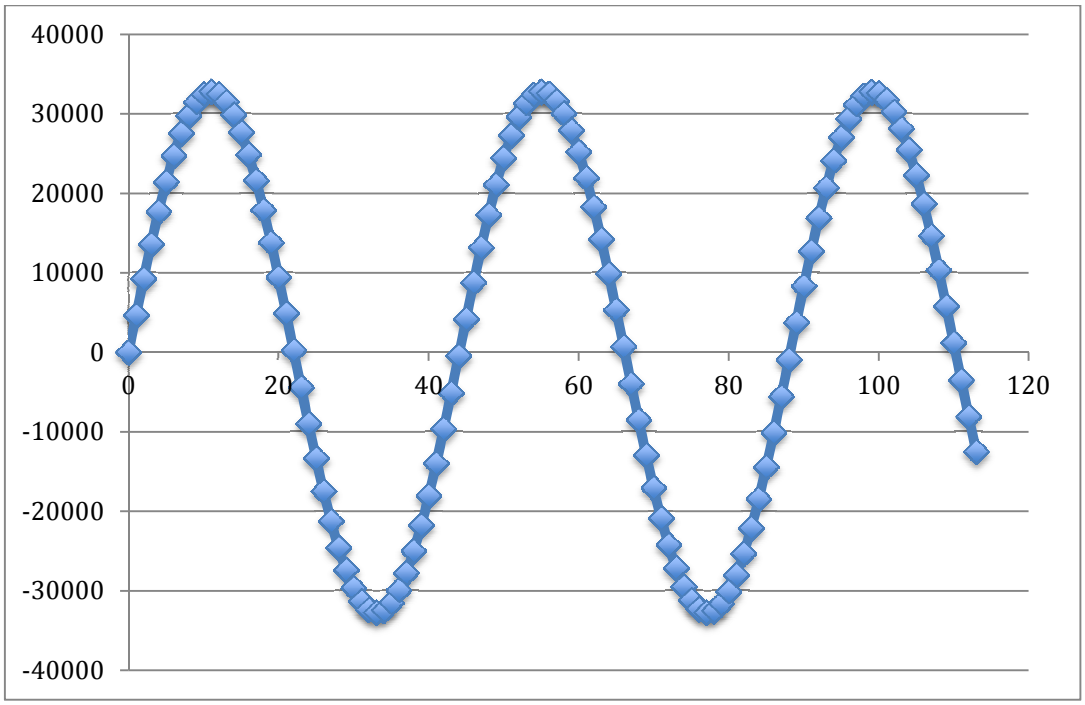

Figure 3: 1,000 Hz Sine Wave Buffer

Understanding Sound Analysis (a.k.a. Using a Fourier Transforms):

In order to receive data, the program needs to read from the phone's microphone. A microphone (Figure 4) is kind of like a reverse speaker. Instead of converting electrical signals into vibrations, it converts vibrations into electrical signals. These signals are then amplified and read programmatically. On an Android system, they are dealt with very similarly to speakers. They have the same number of discrete positions with $(2^{15} - 1)$ being the maximum extension, (0) being the middle, and (-2^{15}) being the maximum retraction. They also are limited to sampling at 44,100 samples per second, which means a maximum frequency of 22.05 kHz. However, when listening for a tone, you cannot specify just one frequency to listen for. All tones are superimposed on one another and only the combination of those tones is heard. The microphone reads the sum of these overlapping frequencies over time; this is known as the time domain. How

does one know when a certain frequency is playing or not? By using what is called a Fourier Transform.

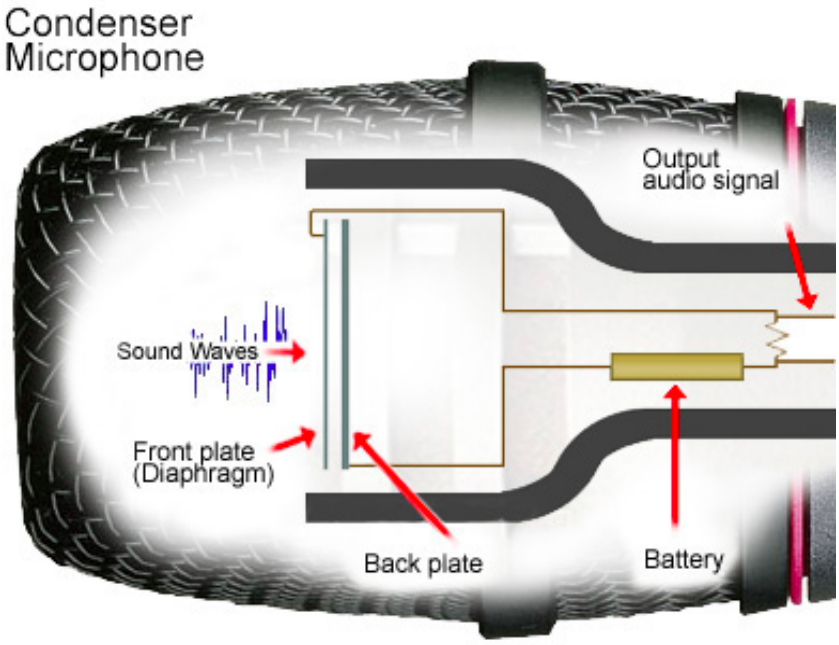

Figure 4: Microphone

A Fourier Transform is a conversion from the time domain (amplitudes over time) to the frequency domain (for a given moment in time: power over frequency). This is useful because the microphone outputs a summation of frequencies, but if the goal is to determine how much power is in the frequency range of 18kHz to 19kHz, a Fourier Transform can be used to find the power spectrum for specific frequencies.

The graph in Figure 5 is a summation of four different frequencies of sine waves over time (aka in the time domain). If a Fourier transform is applied to that segment of time, it can be seen that it is actually comprised of four different frequencies of sine functions, each of which have a constant magnitude of one.

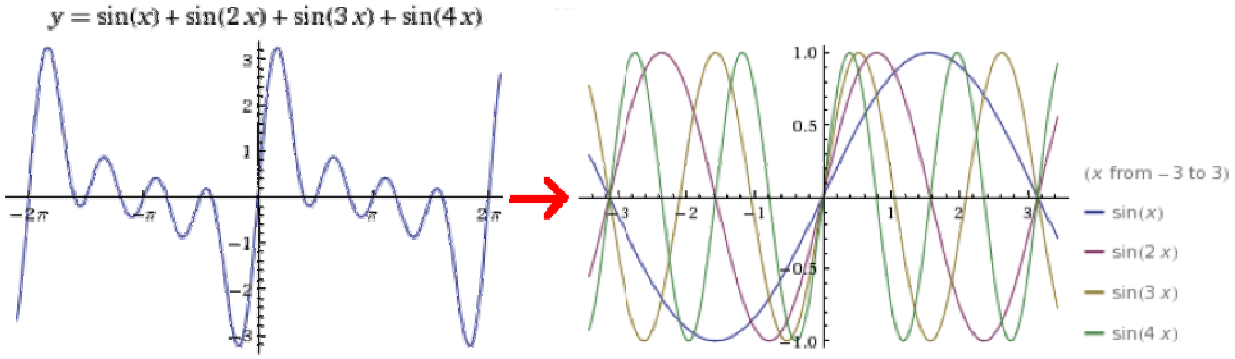

Figure 5: Application of Fourier Transform

Unlike the example in Figure 5, not all Fourier Transforms show that all frequencies have equal magnitudes. In fact, that is almost never the case. Usually, the frequencies are all of different magnitudes and the purpose of the Fourier Transform is to detect those differences. A more realistic example can be seen in Figure 6.

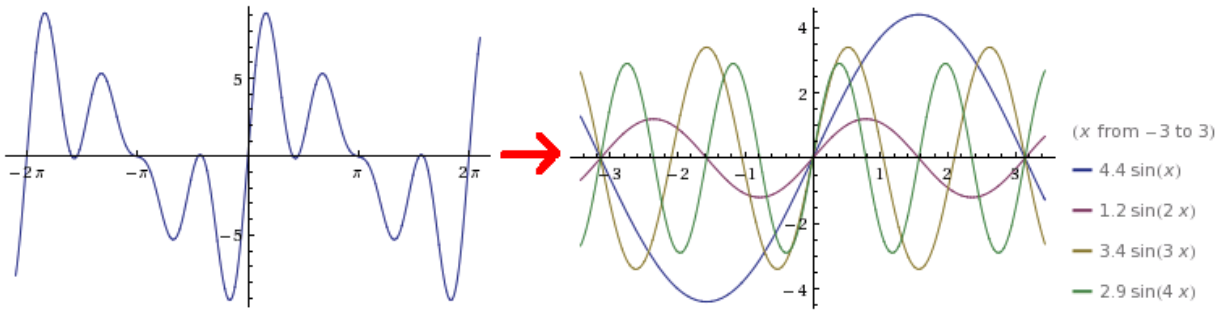

Figure 6: Application of a Fourier Transform of Frequencies of Varied Intensity

In Figure 7, you can see a real Fourier transform of the sound waves in a room. As you can see there is a huge spike at 19kHz. That is because while recording the sample to transform, a 19kHz tone was being played nearby. So it is correct to show that more power was in the 19kHz range than the 18 or 20 kHz ranges.

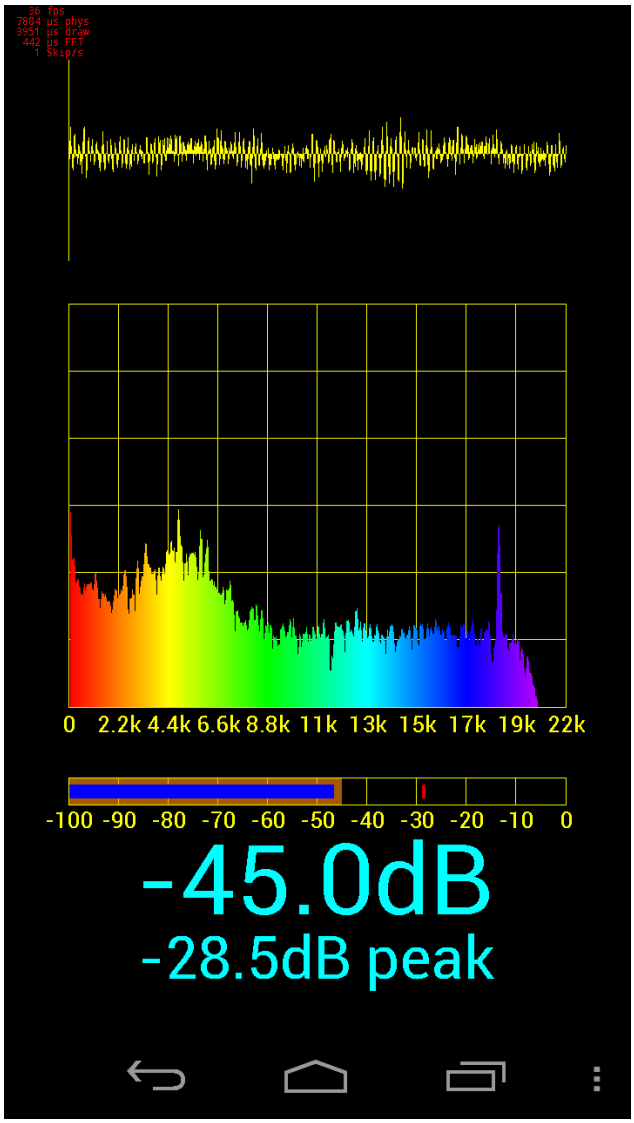

Figure 7: A Fourier Transform of the Room with a 19kHz Tone Playing

Chapter 3: Description

Basic Data Transmissions:

After I was able to generate a wave and perform a real time Fourier Transform, I put it to the test to see if I could send some ones and zeros. I programmed one phone to send a 20kHz tone for half a second, no tone for another half second, and to repeat this pattern over and over again. The other phone was programmed to perform a Fourier transform every half second, and if a 20kHz tone above a specified power threshold is present, mark that as a '1'. If not, it marks it as a '0'. In theory, it should receive a "10101010..." pattern. The purpose of this test was to see if data transfer is possible in the program's current state. After a few tiny adjustments, it worked and I could see the data as it was being received.

Since I could watch the incoming data, I could also see when a glitch would occur. Luckily, it did not glitch when the devices were at a reasonable distance. So I turned up the frequency. This time, instead of a 500-millisecond (mS) delay in between state changes, I made it 100-mS. This also showed great results. I continued increasing the baud rate until I found it to be unreliable around a 10 to 50-mS delay between tones.

Potential Protocols:

Now knowing that data transmission is possible, I needed to decide on a protocol to use. After some careful thought, I came up with three potential approaches.

Solution 1:

The first and most basic solution was to use the same synchronous approach I used earlier as a proof of concept: a 20kHz tone representing a '1' and a lack of a 20kHz tone representing a

‘0’. The phones would need to sync up by some external method, like both users hitting start at the same time, or with an internet clock with a predetermined timer interval. For example, the time interval is 100mS and the sending phone is attempting to send the bit stream ‘10’. Once both phones are synchronized, the sending phone will play a 20kHz tone for 100mS to represent a ‘1’. Meanwhile, the second phone would delay for about 40mS, sample for 30mS to detect the 20kHz tone, and delay again for another 30mS to end that period. Then, the next 100mS, the sending phone would do absolutely nothing so as to send a ‘0’, and the receiving phone would perform the same action as before to find an absence of the 20kHz tone and interpret it as a ‘0’.

This is bit transmission protocol can be coupled with a simple dataflow protocol. The first eight bits would be dedicated to the length of the rest of the transmission (in bits). This is done to let the receiving phone know how long to listen before it interprets the data received. This is a very simple protocol and therefore has the benefit that it is easy to program. Also, since it only spans a single frequency, it is difficult to corrupt and the Fourier Transform algorithm can use a much smaller block size, like 128 or even 64, which is computed much more quickly than a block size of 1024.

However, there are two major flaws with this solution. The fact that only one bit is sent at a time means either the analog baud rate need to be a lot higher, or the bandwidth needs to be lower, and it lacks any kind of error detection. Although, it is possible to implement error detection, it would require the phone on the receiving end to respond periodically, which would drastically increase the complexity of the protocol.

<u>Solution 2:</u>

The second approach involves a much more complex solution. This would involve utilizing ten frequencies simultaneously. Eight frequencies of the transmission would be used in the same manner as the first approach (representing a '1' if the frequency is present and a '0' if it is not), one frequency dedicated as a parity bit, and one frequency used as a clock. The clock would be high, only when the data is ready to be read by the receiver. This eliminates the need for the phones to externally synchronize. However, it still does not eliminate the fact that both phones would need to know who is sending and who is receiving ahead of time.

This method will use even parity and the same synchronous period based timing scheme as the last, except this time both phones will be generating tones and it will be driven by the phone sending the data. The receiver will start out by constantly analyzing all incoming sounds and looking for the clock frequency. As soon as it detects the clock frequency, it will continue performing transforms on the room and recording a running average until the period is over. At the beginning of the second period, the receiving phone will analyze the data and make sure the transmission has an even parity. If it does, it will respond with a 20kHz tone for the rest of the period. If the packet has an odd parity it will not respond at all. Meanwhile, if the sending phone receives the tone, it will consider that packet sent and will continue on to the next packet. If not, it will resend that packet. The parity check will catch 50% of all errors and improve robustness.

Like the previous method, the first transmission will determine the length of future transmissions. But since both phones are generating and analyzing tones in this model, a checksum can be sent at the end of each transmission. The checksum will be used to analyze the data previously transmitted and ensure it was correct. Checksums are not 100% accurate, but they do a good job, especially when coupled with a parity check.

This method drastically increases bandwidth by a factor of eight. This method also increases robustness with its multiple error checking methods. However, it severely increases the complexity of the program and the dataflow protocol. Also, each transmission is more prone to errors and there is wasted time during each acknowledgement.

Solution 3:

The third solution is similar to the second, with a few key differences. The first difference is that this solution is asynchronous (a.k.a. does not use the idea of the timing intervals). Instead, when the device sends the data it will continue to send it over until it receives an acknowledgement from the receiver. This allows the receiver to take as much time as it needs to analyze the signal and make sure it passes the parity check. The frequency the receiver uses to respond needs to be reserved for only the acknowledgements; otherwise, if the sender were driving that frequency high as well, it would not be able to tell the difference between the frequency it is generating and the frequency of the receiver's acknowledgement. The second difference is the necessity to free up one of the ten bits used in Solution 2 since there are a limited number of frequencies available. Most data used in computers these days are in logical bytes of eight, so it would be unwise to transmit seven at a time, and there's no point of sending an acknowledgement that the parity matches if there is no parity bit to begin with. That means clock line must be eliminated. Instead of triggering on the clock, the receiver can trigger as soon as it recognizes any of the nine frequencies playing. By doing this, a line can be reserved solely for the acknowledgement, but there is still an issue. If the sender tries to send a full byte of '0's the parity bit will be a '0' as well to keep the even parity. To mitigate this issue, the parity needs to change from even to odd. Now, this edge case will no longer exist because there will always

be an odd number of ‘1’s transferred, which means there must be at least one of them. As soon as the acknowledgement, the sender will pause for a short period of time (without playing any tones) to notify the receiver that it received the acknowledgement and is now switching to the next byte of data. This pause is essential because if it didn’t exist and the sender sent two identical bytes sequentially the receiver will be unable to detect the second byte.

This method will render a higher baud rate than the first, and maybe even the second, since it will not waste time resending bytes due to incorrect parity or transmitting nothing but acknowledgements. It also makes the parity error checking less complex and more robust. Furthermore, the protocol does not have to maintain a constant synchronization, so the garbage collector of the Java virtual machine will not interfere with the timing. The only down side is that the data is not transmitted with a regimented timeframe, which means it will be difficult to guarantee a low latency game that relies on this protocol. Also, the devices need to be capable of simultaneously playing and analyzing tones.

Note: All of the above methods share the major problem of needing to know ahead of time whether, they will start by sending or receiving data. This limits the devices to half duplex at best. This is solvable by first implementing a discovery mode, followed by a handshake, and a clever usage of time division multiplexing, however I do not cover that protocol in this paper.

Deciding on a Protocol:

Before I could decide on a protocol, I had to fully understand the environment that I created. Specifically, this meant understanding all the timing constraints and hardware limitations.

Each potential solution has different strengths and weaknesses, and their performance is dependent on different aspects of the environment in which they are implemented. For example: if the major limitation of the environment was the analog baud rate, I could use a method that delivers eight bits at a time using, with a baud of only ten, which would deliver up to 80 bps. However, this would put major strain on the generation of the tone to be delivered. Since this seemed like the most feasible approach at first, I had to make sure I could generate a tone with more than one frequency in it, then set up a test to see how long it takes to generate those tones.

Generating Tones with Multiple Frequencies:

Many devices in the world today have the ability to simultaneously play multiple audio tracks. For example, if one were to visit both YouTube.com as well as Pandora.com, they could experience both watching and listening to a clip on YouTube, with the background music from Pandora. This same concept is possible on Android devices that are capable of multitasking. So I knew it was possible for this project.

To play multiple samples on the speaker simultaneously on the Android platform, multiple instances of the AudioTrack object must be initialized in their own thread. This would require a total of ten threads, one for each frequency, to be continuously running. Then, all ten of them would have to have access to some shared memory which would be used to notify them when it is appropriate to start playing and when to stop. But having ten threads attempting to control one piece of hardware for tiny intervals did not seem like the best idea, so this approach was left as a backup plan.

There are two methods to ensure mutually exclusive access to the speaker: using time division multiplexing or finding a way to play all the frequencies simultaneously from a single

buffer. Time division multiplexing requires playing one tone, then quickly switch over and playing another, and then quickly play the next, etc. This approach would actually work; however, the speaker would have to sample much faster than the microphone, and since they have the same sample rate it is not a great solution.

The only way to effectively do this is to superimpose all the tones into one buffer. There are four steps to do this with the most precision:

Step One: Create a buffer of doubles to hold the double result of a Math.sin() computation.

Step Two: Combine the results of each waveform individually into that buffer.

Step Three: Scale every element down by a factor corresponding to the total number of frequencies combined in the buffer. This is done to keep the range of the combined sine values between -1 and 1. (In other words, if three frequencies are combined by summing up the three buffers into one, every element in the buffer needs to be divided by three so that none of the values are out of the range [-1.0, 1.0])

Step Four: Convert the buffer of doubles into a buffer of shorts by multiplying the normalized values by the max volume (max volume == $2^{15} - 1$ == Short.MAX_VALUE.)

Step three and four should be performed simultaneously to increase performance, but they are broken up to make this explanation more clear. This method actually works! But after testing it for a while, I realized the volume dropped dramatically. This happened, because when scaling the values, it was assumed there existed a point where all the frequencies overlapped at their highest point. This is probably not going to be the case. So to attain the maximum volume (which extends the transmission range), the buffer needs to be scaled as little as possible. Luckily, this is easy because the maximum point in the double buffer is not to exceed 1.0. All

that needs to be done is iterate through the buffer and find the maximum value, then scale the rest of the buffer by that same value. This allows the speaker to play multiple tones from one thread with the maximum volume possible.

Experimenting with Tone Generation Performance:

The first few tests showed that it took up to 99mS to generate a tone 100mS tone comprised of ten frequencies. It would not be feasible to spend only 50% of the time actually sending data with a guaranteed 50% minimum overhead. So I was left with three options at this point: optimize the waveform generation, pre-generate all the waveforms, or choose a different method.

At first, I went through and optimized the waveform generation algorithms. I was able to use the specific FloatMath.sin() function and allocate all the memory at once, which drastically reduced garbage collection time; however, it still wasn't fast enough.

Although pre-generating all the waveforms seems like it would be impossible because of the memory requirements, after doing a few simple calculations, it turns out that it really isn't that much memory since the duration of the buffers are so small. When using a 100mS waveform, each waveform will be an array of 4,410 shorts and a unique waveform is necessary for every bit combination of the ten bits, which is 1,024 waveforms. This means it would take up about:

$$1024\ waveforms * 4410\ \frac{shorts}{sample} * 2\ \frac{bytes}{short} \approx 9\ million\ bytes \approx 9\ megabytes$$

Considering most phones have around 256 to 1,024 mega bytes of ram these days, 9mb is not a problem at all. Now, with all the tones pre-generated, the only time it takes to play them is the time it takes to index into the array.

Clicking Issue:

There is still one issue with the tone generation at this point. There is a noticeably loud clicking sound at the beginning of some of the tones. At first, I thought this was due to the AudioTrack software initialization; however, I quickly realized that even if a single buffer contained several tones back to back, the click still occurred at the beginning of each tone. After playing around for a little while, I realized that the issue was only happening on tones that consisted on combinations of more than one frequency. One of the tones that made a significant clicking sound was one that consisted of the following frequencies: 19.6kHz, 19.4kHz, 19.2kHz, 19.0kHz, 18.8kHz, 18.6kHz, and 18.4kHz. After numerous attempts at a solution, I decided to graph the sample (Figure 8).

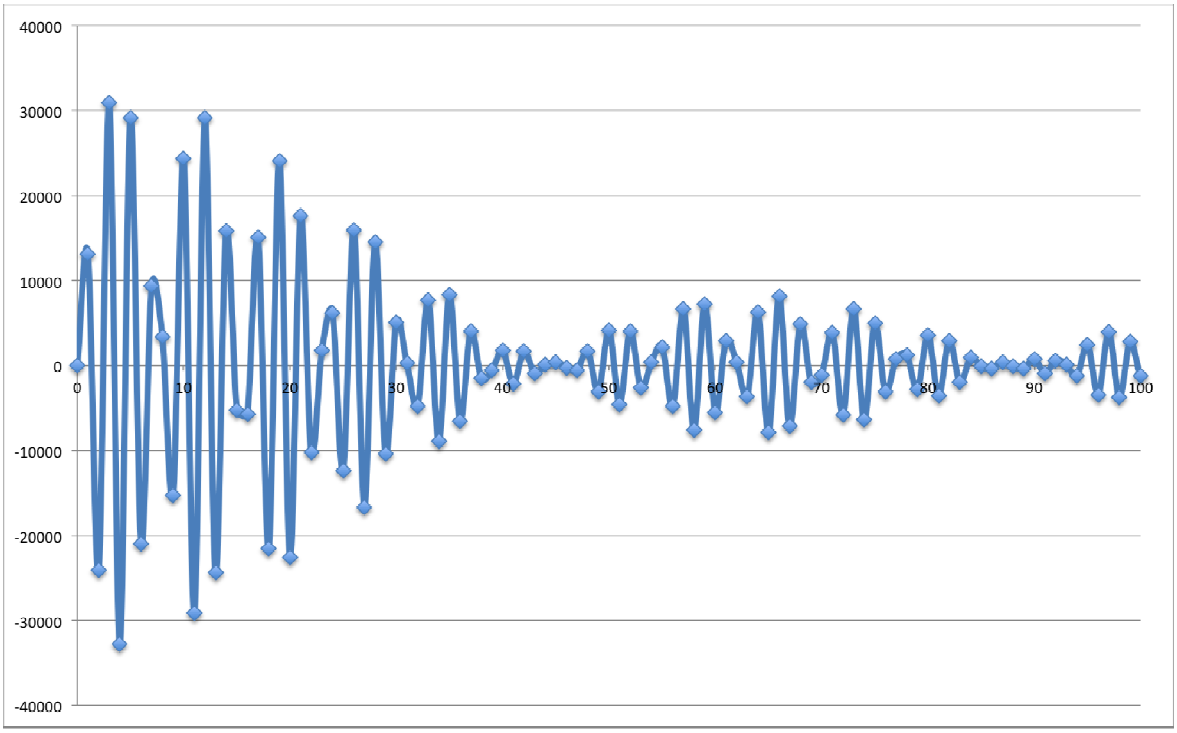

Figure 8: 100 Samples of a Waveform Comprised of Seven Frequencies

Everything looked pretty normal. It does look like there's an abrupt start to the waveform though. But I couldn't see the period, so I graphed a larger sample (Figure 9).

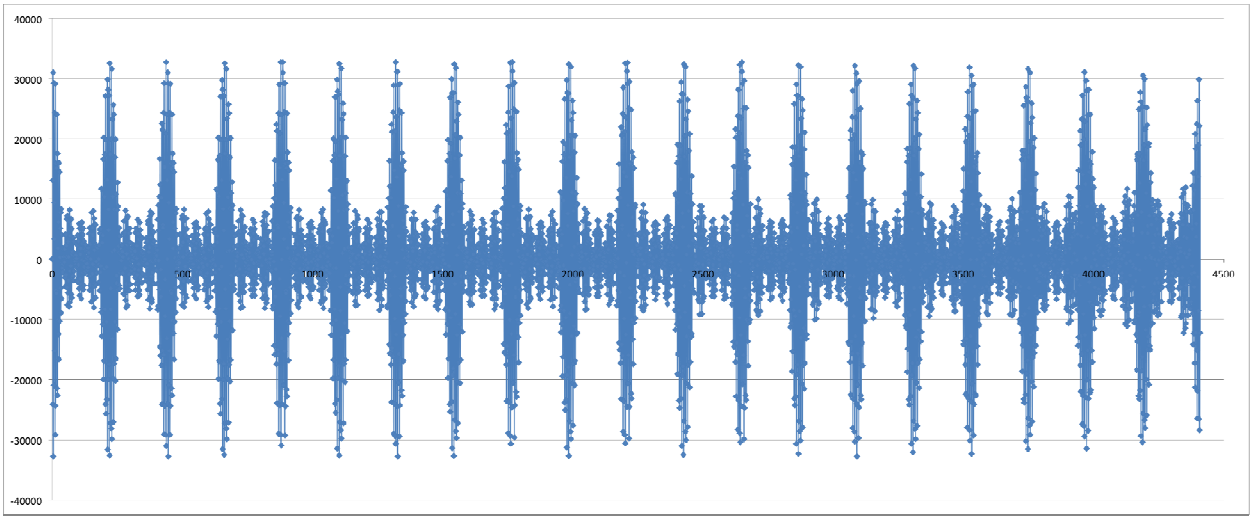

Figure 9: 44,100 Samples of a Waveform Comprised of Seven Frequencies

After that, the problem seemed obvious. The waveform starts and ends in the middle of a peak. That means, the speakers start from position zero and then abruptly jump to the highest point in the middle of a peak and that's what is causing the clicking. To solve this problem, all that must be done is start the waveform generation somewhere at the bottom near zero, so the initial jump doesn't need to be as abrupt. This is done using a peak detection algorithm that looks for the first trough after the largest peak in the waveform and starts and stops the sample in those areas.

The Protocol to Use:

Now, considering the devices have the ability to play tones comprised of multiple frequencies, as well as the ability to read and write at the same time, the clear choice is to use the asynchronous solution (Solution 3). This will greatly simplify error checking, increase robustness, and it could potentially speed up the transmission speeds.

Implementing the Framework:

The system is laid out in a relatively simple manner. The main program is called Wave. This class contains all of the functions necessary to send and receive data. After the object is constructed, all you need to do is call Wave.receive(byte data[], int timeout) on one phone and then Wave.send(byte data[], int timeout) on the other (the order in which they are called do not matter because of the asynchronous manner of the protocol). The Wave class relies on two other classes, AudioGenerator and AudioAnalyzer.

The main purpose of the AudioGenerator class is to facilitate the ability to create and play tones with as low latency as possible. When constructed, it allocates enough memory to contain every possible tone and every possible tone combination the user can request. With ten different frequencies, there are 1,024 different tones. These sine waves need to be as accurate as possible, so as one might have guessed, this initial generation time is rather large, especially considering all the complex calculations necessary to combine different frequencies and avoid the "clicking issue" noted earlier. Since this takes a while, it is done in a background thread. (See conclusion for more details on how to fix the speed of generation.) Then once the background thread is done generating the waveforms, the thread drops off into an infinite loop that waits for a boolean "playing" to be set to true. A class function AudioGenerator.play(int waveform) accepts an integer [0, 1023] and sets a local variable "tone" to that value as well as sets "playing" to true.

This triggers the infinite loop to continuously write the waveform corresponding to “tone” to the speaker as long as “playing” is still true. AudioGenerator.stop() simply sets “playing” to false, and flushes the additional audio buffer that was queued for playback.

The other class Wave utilizes is AudioAnalyzer. This class is responsible for reading the data from the audio input buffer and analyzing it by performing a Fourier Transform using a ManualAnalyzer class written by Ian Cameron Smith. It does this by running a background thread that constantly performs the Transform on the audio in the room and places the results in a synchronized variable. That way, whenever the program needs a Fourier Transform, it is already done and it just needs to be referenced. When it is referenced, certain key blocks are compared against an average of the room’s ambient sounds, and this comparison lets the program know whether or not the frequencies are present. So this all sums up to a simple function AudioAnalyzer.listen() which returns an int corresponding to the ten data bits.

Protocol Details:

Below is a flowchart of how WaveChat uses the Wave class to send a byte of data and Figure 10 shows the spectral sonogram of the text “Hey” being sent from one phone to the other:

Time	Phone A (sending)	Phone B (receiving)
T=0		User hits receive and phone waits to receive data by listening for any data bits 0 through 7 and bit 8 for parity. (Bit 9 is reserved for acknowledgements or ACKs for short)
T=1	User hits send and data starts playing continuously on bits 0 through 7 and bit 8 for parity, and begins listening for an ACK on bit 9. (At this point it should not be present)	

T=2		Phone B hears data bits present and continues to take a few more samples to have a good average. At this point Phone B will make sure the parity bit matches. If it does, it will then drive bit 9 high to signal an ACK. If it is not, it will start the process over again until the parity bit matches.
T=3	Phone A will recognize the ACK and completely silence its speaker. Then it will keep monitoring the data lines until Phone B stops driving the ACK signal.	
T=4		Once Phone B recognizes that Phone A has stopped playing data, Phone B will drop its ACK signal and begin waiting for the next data packet.
T=5	Phone A will recognize the complete silence and will be aware that the last transfer is complete and that Phone B is ready for the next packet.	

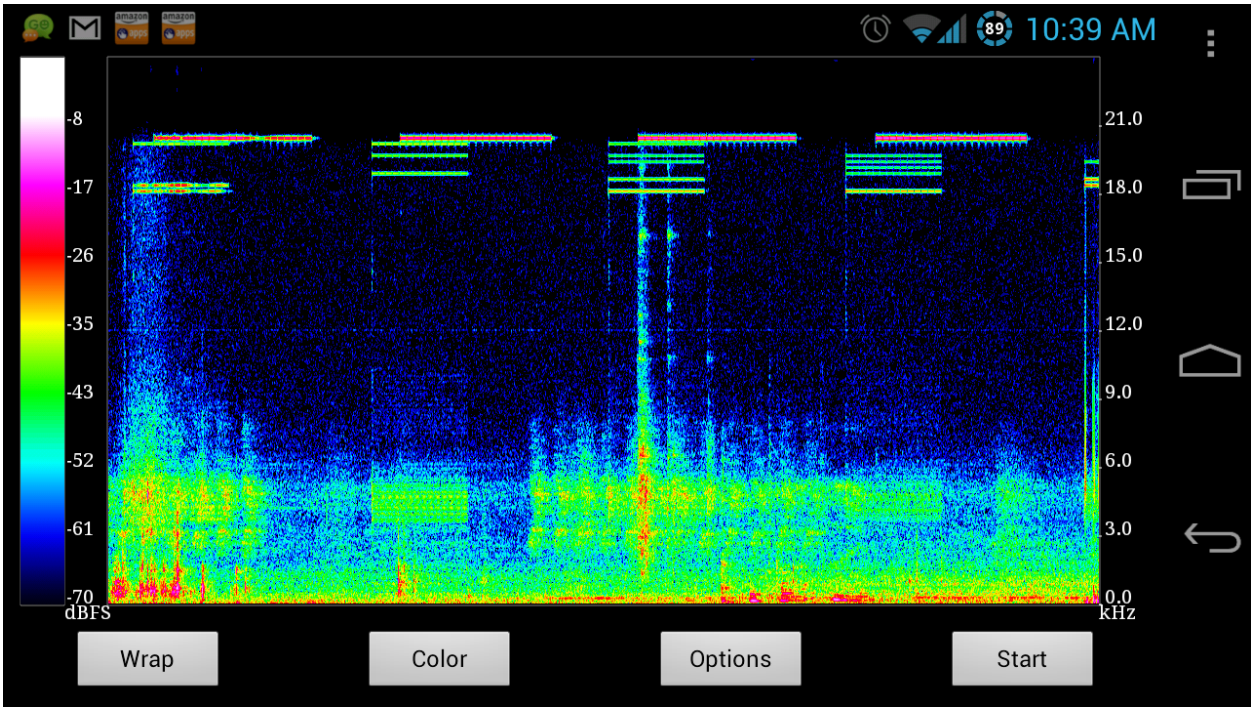

Figure 10: The exchange of the text "Hey" between two devices

Chapter 4: Evaluation

In order to evaluate the deliverables, I used the following series of requirements and tests:

Simplex Data Transfer Test:

Requirement: The device must be able to send data from one phone to the other.

Test Case: Send a string of text from one phone and verify that it is received on the other end.

Result: Pass. It is possible to send data from one phone to the other.

Distance Test:

Requirement: The devices must be able to communicate within a distance of at least five feet. (If it were any less, the two players might as well just share a screen!)

Test Case: Perform the Data Transfer Test between the phones at a distance of zero feet (the phones are touching). Continually increase this distance by half a foot and repeat the data transfer test until the packet drop rate is higher than 50%.

Result: Fail. There is max range of one foot. This is due to the frequency bleeding which limits the volume of each packet, which in turn, limits the distance.

Data Rate Test:

Requirement: The data must be able to be transmitted at a reasonable data rate. However, given this is to run on a relatively weak (based on today's standards) machine and it is going to be through a medium which has a lot of noise, and a "reasonable data rate" is not

at all impressive on paper. So I would set the pass criteria to be 16bps, with a goal of 80bps.

Test Case: Send a large amount of data and time how long it takes for the file to be sent, this will determine the max data rate.

Result: Fail. The resulting data rate is closer to speeds of 4 bps, which is way slower than expected. This is entirely due to the high latency of the Android audio drivers; however if desired, it could be worked around by using a different protocol.

Half-Duplex Test:

Requirement: The data transfer must be at least half duplex. Full duplex would be great and is probably possible to some degree. Although, it is probable that implementing the full duplex capability would drop transfer speeds from phone A to phone B, but would increase overall data transfer over time (a.k.a. data from A to B + data from B to A).

Test Case: Send a string of data from phone A to phone B and upon receiving it, respond with a unique response from phone B to phone A. This will confirm or deny the ability to send data in both directions.

Result: Pass. The phones are both able to send data when it is their turn. However, since there is nothing specified in the protocol as of yet and there are no buffers, their "turn" is decided by when one phone pressed receive and the other presses send.

Error Detection/Noisy Transfer Test:

Requirement: The device must perform error detection, via a checksum or any other measure, and auto correct if erroneous data is received.

Test Case: Send a packet in a noisy environment to see if the error detection algorithms work well sufficiently.

Result: Pass. While the reliability of the data transfers is very low and there are many errors, the error detection is very accurate; there just isn't much you can do about it.

Chapter 5: Conclusion

Improvements and Future Plans

In its current state, the program was not able to pass all of the evaluation criteria I set out in the beginning of the project. However, that's not to say it is not possible. There are a few key issues that once addressed will adhere to most (if not all) of the criteria that were not passed.

Frequency Bleeding:

One of the most critical issues that I encountered was the concept of frequency bleeding. This is due to the speaker driver's lack of resolution and inability to write to the speaker faster than 44,100 times a second. When a frequency is playing at a given frequency, let's say 18kHz, the speaker actually outputs other frequencies around 18kHz ranging from around [16kHz, 19kHz]. Although the intensities of these spurious tones are significantly lower than that of the intended 18kHz tone, they still cause complications. For example, phone A hears an ACK whenever it plays data since the speakers are so close to the microphone that the bleeding frequencies are higher than the threshold. To mitigate this issue, another peak detection algorithm should be implemented to find if the key frequency intended to be detected is actually higher than the average bleeding frequencies around it. This would greatly increase accuracy without taking a major performance hit.

High Latency:

There are major problems with the Android's ability to read and write data to the audio hardware quickly, delaying roughly about 100-200 mS for each access. The easiest solution is to

switch to a different platform, like iOS, which has a latency of about 5-7 mS. Another method would be to use a protocol that does not require the phones to acknowledge every byte read. Instead, the first phone would send streams of bytes and the second phone would simply record the data coming in until the data stopped. Then the second phone would take its time analyzing the data that came in and respond when it was ready. This way, a number of bytes could be transferred with a single 200 mS offset from the latency, instead of repeated 200 mS offsets every time one phone needs to start playing data or send an ACK back.

High Initialization Time:

Since the program has to generate all the waveforms when it opens, it takes a long time to initialize. One method of cutting back on this time is to serialize all the waveform arrays and save them to an external file, which comes with the program as a resource. Then every time the application boots up, it just needs to populate the arrays with that data instead of regenerating them. This would cut back the 100 second delay by more than 99 seconds.

Speaker Placement and Propagation Delay of Sound Waves:

One issue is that not all phones have uniform speaker and microphone placements. This makes it difficult to code for edge cases where they are muffled or right next to one another. Another issue is the actual propagation delay of sound through dry atmosphere is about one millisecond at one foot. That alone introduces latency problems. Sadly, there is nothing to be done about these two issues. Unless we buy all cell phone manufacturers and implement a new standard, and change the density of the air, we're just going to have to live in the constrained environments we are given and accept the fact that it will not be the best method available.

Overall:

The program worked to some degree. Even though the program is unreliable, it can be gathered from the project that this method of data transfer is entirely feasible. However, due to the extremely low bandwidth, the idea that this could be used for real time multiplayer gaming is impractical. Even with the previously mentioned improvements to increase the bandwidth, there is still the major problem of the high latency, which hinders any possibility for real time gaming. However, that does not leave this program without use! There is great use for a simple protocol like this one. For example, it can be used to automatically pair two phones via Bluetooth with the push of a button. The Bluetooth could then be used for the real time gaming. Or, better yet, this program could be used to share one phones WiFi information with surrounding phones, so they could all easily connect to the network with the push of a button, instead of having everyone exchange a ten digit hex password.

Overall, this project tested my skills as a computer engineer on a number of levels. On the hardware side: I had to learn to incorporate multithreaded access to system resources like the speaker and microphone, convert digital data into mixed analog signals and vise versa, as well as learn as the physical properties and limitations of speaker and microphone hardware. On the software side: I learned to write peak detection algorithms, audio generation, synchronous and asynchronous data transfer protocols, error detection and correction. I also learned how to use a Fast Fourier Transform effectively. After integrating a wide variety of different engineering technologies, I was able to prove that the concept is possible. Even though the project is not to be used as originally intended, I would still consider it a success.

Annotated Bibliography

"Experimental Feature." Wolfram|Alpha: Computational Knowledge Engine. Wolfram Research Company. Web. 18 May 2012. <http://www.wolframalpha.com/>. Used to create graphical representation of data both collected and generated.

Harris, Tom. "How Speakers Work." HowStuffWorks.com. HowStuffWorks. Web. 18 May 2012. <http://electronics.howstuffworks.com/speaker10.htm>. Used image from this website as a base to create my own speaker image.

"How Sound Waves Work." Sound Waves. Wavelength Media. Web. 05 Mar. 2012. <http://www.mediacollege.com/audio/01/sound-waves.html>. Outlines a description of the fundamental properties of waves and, more specifically, sound waves.

"Microphones: How Microphones Work; Types of Microphones; Choosing a Microphone." Microphones: How Microphones Work; Types of Microphones; Choosing a Microphone. AudioLink Services. Web. 18 May 2012. <https://microphones.audiolinks.com/microphones.shtml>. Used for microphone image.

"Newton's Cradle Classic Megamind Edition." Apple - App Store. Web. 18 May 2012. <http://itunes.apple.com/us/app/newtons-cradle-classic-megamind/id288217026?mt=8>. Used for image of Newton's Cradle.

"Reference." Android Developers. Google. Web. 05 Mar. 2012. <http://developer.android.com/index.html>. Everything one needs to program for an Android device, including the SDK, documentation, examples and more.

Smith, Ian C. "Introduction." Audalyzer. Web. 18 May 2012. <http://code.google.com/p/moonblink/wiki/Audalyzer>. Used for Fourier Transform Library.

Weisstein, Eric W. "Fast Fourier Transform -- from Wolfram MathWorld." Wolfram MathWorld: The Web's Most Extensive Mathematics Resource. MathWorld--A Wolfram Web Resource. Web. 05 Mar. 2012. <http://mathworld.wolfram.com/FastFourierTransform.html>. This source outlines the idea behind the Fourier transform and basics of how to implement the FFT (Fast Fourier Transform) algorithm.