EAC, GTH
Project Number:EXC-0901

Time Synchronization in 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

Date: December 22, 2009

Approved: Professor Edward A. Clancy

Professor George T. 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 method to determine the position of a sound source via an array of acoustic sensors. In an earlier project, entitled Acoustic Localization for Mobile Open-Source Network Deployment (ALMOND), an attempt was made to develop a system for acoustic localization using cell phones as acoustic sensors [ALMOND, 2009]. A prototype system was developed, although it was capable of performing only rudimentary estimates of localization, and thus was not suitable for a real world application. One of the most pressing obstacles which had not been overcome was the ability to precisely determine the time at which audio samples were recorded from each phone's microphone. This project was able to reduce the error in determining when individual timestamps were recorded by modifying techniques of retrieving timestamps and filtering the results by applying moving medians and moving means to the differences between successive timestamps. Using these methods, differences between successive timestamps always fell within three milliseconds of expected values and fell within one millisecond of expected values 99 percent of the time.

Acknowledgements

Much of the code and motivation for this project was created during an earlier project entitled Acoustic Localization for Mobile Open Source Development [ALMOND, 2009]. The author would like to acknowledge the contributions of two of the authors of the ALMOND project: Daniel Skehan and Muhammad Saleem.

The author would also like to thank his advisors at Worcester Polytechnic Institute - Professor Edward Clancy and Professor George Heineman - and his sponsors at Lincoln Laboratories - Albert Reuther and Glenn Schrader.

Table of Contents

Abstract 2

Acknowledgements 3

Introduction 5

State of the Previous Acoustic Localization Project at its Conclusion 7

The State of Time Data Collection and Synchronization 9

Retrieving Precise Timestamps from an Individual Phone 9

Synchronizing the Time Between Two Phones 15

Obtaining More Precise Timestamps Through ALSA API 18

Confirm All Audio Data are Being Recorded 18

Obtaining Timestamps From ALSA 22

Methodology 22

Results and Discussion 23

Reducing the Priority of the Phone's Program when Using ALSA Timestamps 29

Modifying Timestamps to Improve Precision 32

Moving Averages 32

Moving Medians 35

Synchronizing Time Between Phones 37

Methodology 37

Results and Discussion 39

Discussion 41

Conclusions 43

References 46

Appendix A: Program to Retrieve and Record Timestamps using ALSA API Function Calls 47

1. Introduction

Acoustic localization is a method to determine the position of a sound source via an array of acoustic sensors, as shown in Figure 1. In an earlier project, entitled Acoustic Localization for Mobile Open-Source Network Deployment (ALMOND), an attempt was made to develop a system for acoustic localization using cell phones as acoustic sensors [ALMOND, 2009]. The purpose of the project was to take advantage of the wide spread availability of cell phones and numerous features such as wireless communication through Wi-Fi, Global Positioning System receivers, and a microphone. A prototype system was developed, although it was capable of performing only rudimentary estimates of localization, and thus was not suitable for a real world application. One of the most pressing obstacles which had not been overcome was the ability to precisely determine the time at which audio samples were recorded from each phone's microphone.

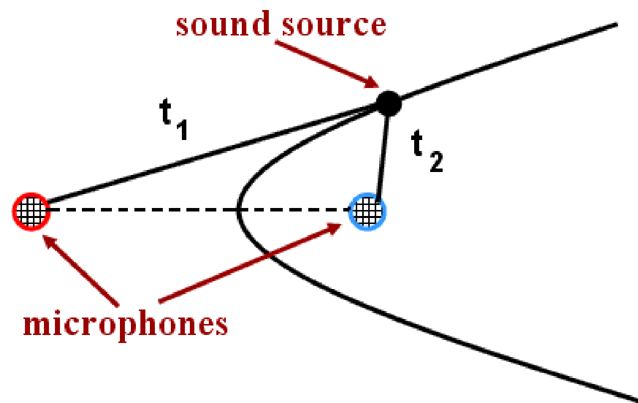


Figure 1 Acoustic Localization with Two Sensors and Derived Hyperbolic Curve (Modified from [Birchfield and Gillmor])

In the ALMOND project, acoustic localization was based on a time difference of arrival approach. The prototype system was composed of two phones and a sound source that generated audible chirps. When two phones are used, the location of the sound cannot be uniquely

determined. Instead, when a sound was detected by both phones, the system should be able to compute a hyperbola of possible sound source locations. If a third phone were added, three hyperbolas could be drawn, one for each combination of two phones; the intersection of these hyperbolas would be the planar computed location of the sound source. This method is highly dependent on being able to precisely determine the time at which a sound reaches each phone. The speed of sound can be approximated as 340 meters per second at sea level. At this speed, with two phones one millisecond of error in the time of arrival measurement will result in a minimum of 1.7 meters of error in the location of the hyperbolic solution. The ALMOND project was able to record sound detection times with a precision of approximately ten milliseconds between a pair of phones, and maintain this precision for several minutes at a time. However the methods for querying the time were less than optimal, as they depended on running a program on the phones at a high priority and the resulting times were only correlated with the correct times.

This current project extends ALMOND by investigating two specific questions:

1.) Is it possible to improve the precision when querying the time data of the phones?

2.) Even if the precision cannot be substantially improved, can we use multiple individual time queries together to estimate a more precise time for any given audio data?

The motive behind these questions is simple: increased precision in time would lead to smaller errors in localization.

2. State of the Previous Acoustic Localization Project at its Conclusion

In the Acoustic Localization for Mobile Open Source Deployment (ALMOND) project, two Openmoko Freerunner cellular phones were used as audio sensors. These phones ran a modified version of the Debian GNU/Linux operating system. The Freerunners were chosen because they were readily available and are open source. For this project, it was determined that implementing an algorithm to detect sounds on each phone would be too difficult with the Freerunners, as the ARM processor they have lacks floating point hardware. Therefore, the phones streamed audio data across an 802.11g wireless router to a Dell Inspiron laptop which served as a central processing point. The general hardware setup can be seen in Figure 2.

Figure 2 – 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]).

The processing performed for sound source localization can be divided into five distinct tasks:

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

Figure 1

2) Recorded sound data are streamed over a wireless network.

3) At a central processing point (the Dell laptop), the data from each phone are analyzed to detect events.

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. For the two phone test system, the central processing point actually calculates a hyperbola of possible locations as explained previously.

An overview of this process can be seen in Figure 3.

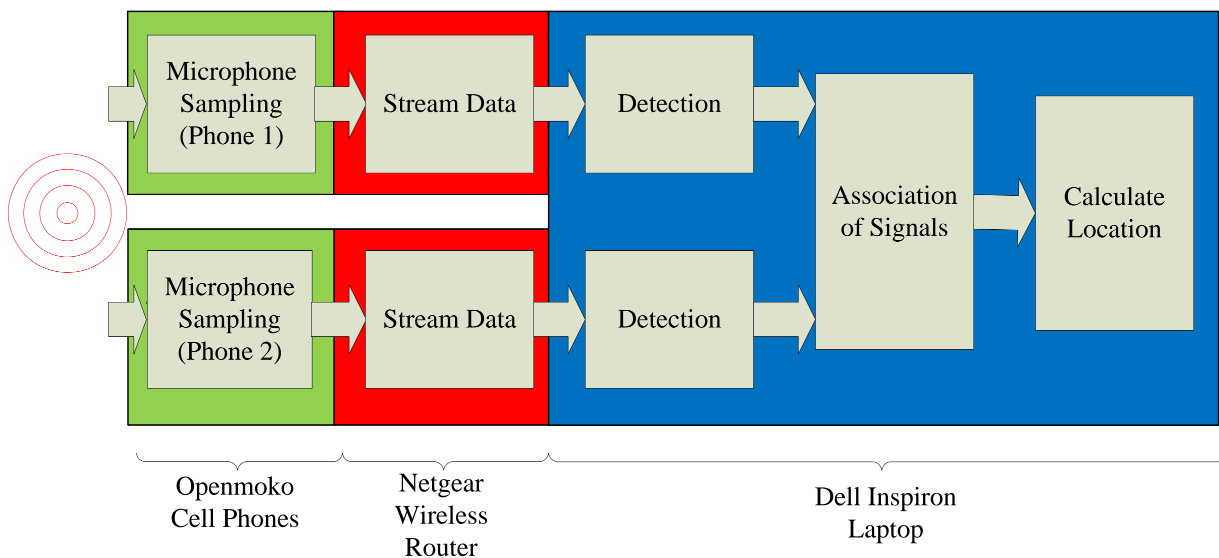


Figure 3 – Block Diagram of Acoustic Localization Method Using Cell Phones as Sensors.

For audio events to be detected, the ALMOND project used chirps (played on a computer controlled speaker) which are sounds that increase in frequency over time. Chirps in a low noise environment were reliably detected using cross correlation with an ideal chirp. This method

provided detection time errors well under one millisecond. For further details, see [ALMOND, 2009].

2.1. The State of Time Data Collection and Synchronization

The problem of time synchronization between the two Openmoko devices in the ALMOND project can be defined in two ways. The first definition is the ability of an individual phone to report a time that corresponds with a specific audio datum. That is, if a phone hears a sound can the phone accurately determine the local time (the time according to that phone's clock) at which it occurred? The second problem is synchronization between the two phones. At any given moment the clock times on the individual phones should be approximately the same (or any differences known and accounted for). It is important to note that the two phones need to read time precisely when compared to one another, but not accurately to any universal time.

2.1.1. Retrieving Precise Timestamps from an Individual Phone

Due to time limitations, the ALMOND project implemented a less than optimal solution for retrieving time information on an individual phone. The basic software loop of phone operation can be characterized by the following four steps (for a more detailed overview of the program, please refer to section four of [ALMOND, 2009]):

1.) Determine the time

The time was queried using the `gettimeofday` function from the glibc libraries. The function call would result in a structure containing the number of seconds and microseconds since January 1, 1970. These data were queued to be sent over the network (the queuing process will be described in the second step). As noted above, this timestamp had to be associated with a data sample. In the ALMOND project, timestamps were associated with the first sample in a buffer of audio data.

2.) Queue data to be sent to the central processing point

To avoid unnecessary system waits which would result in poor quality timestamps, both time and audio data were queued to be sent over the network.

3.) Attempt to Send the Data

The program would, attempt to send the data, but if this would result in the process being blocked (due to network congestion or other factors) then move on to the next step. This non-blocking behavior can be achieved by using the glibc function `send` with the `MSG_DONTWAIT` flag. If the call would block, `send` returns an `EAGAIN` error. If the call would not block, it would send as much data as possible. The program would move to the next step when either the call would block or when there were no data left to be sent.

4.) Record audio data

Audio data were recorded using calls to the Advanced Linux Sound Architecture (ALSA) Application Programming Interface (API). ALSA operates by default using a ring buffer. The sound card on the phone will constantly sample audio data and store it in memory. When requested, ALSA will provide a user with a buffer of the least recently recorded audio samples which have not yet been provided to the user. These data were placed into a buffer which was put into the queue to be sent during the next iteration through the loop.

The problem with this method has to do with how the ALSA internally recorded audio data. An API call to retrieve audio data will return audio data that was recorded before the call if such data are available due to the ring buffer. The general assumption used when retrieving the time was that it corresponded to the first audio sample in the next buffer and that executing the rest of the loop took a negligible amount of time. Unfortunately, this is not always the case.

One way to measure the precision of the timestamps is to compare subsequent timestamps on the same phone. If we know the sampling frequency and we know the size of each buffer of audio data then the time difference between timestamps would ideally be a constant (and predictable) value. In our tests, the sampling frequency was 44,100 hertz, and the size of the buffer was 1024 audio samples. Thus, the time between timestamps should be 23.22 milliseconds. By recording the timestamps and subtracting the ith timestamp from the $(i + 1)^{th}$ timestamp, we can plot a histogram of the time differences. This test was rerun at the beginning of this project to verify a working system and the results are comparable to the ALMOND project. Histograms for both phones for this test can be seen in Figure 4.

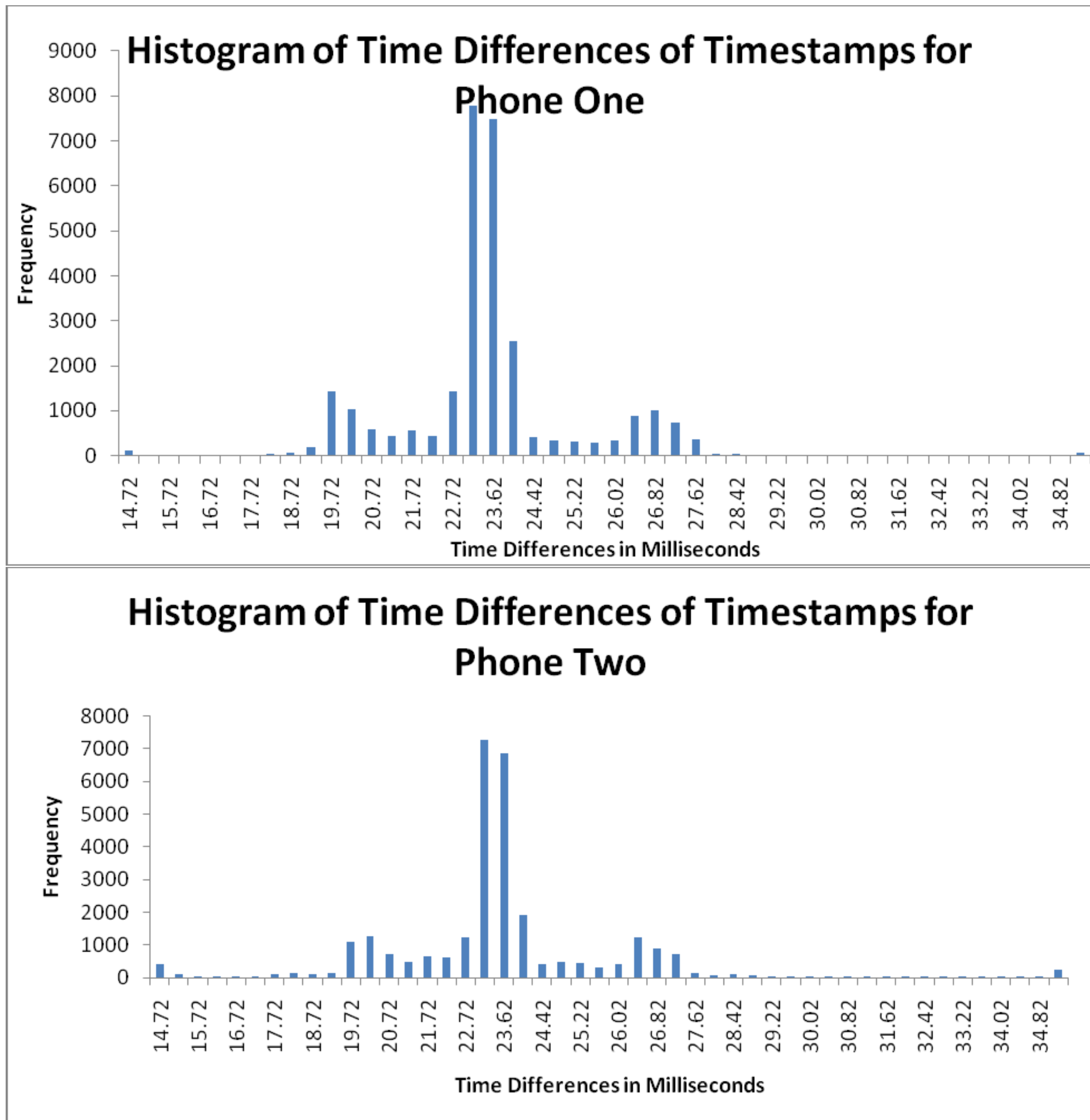
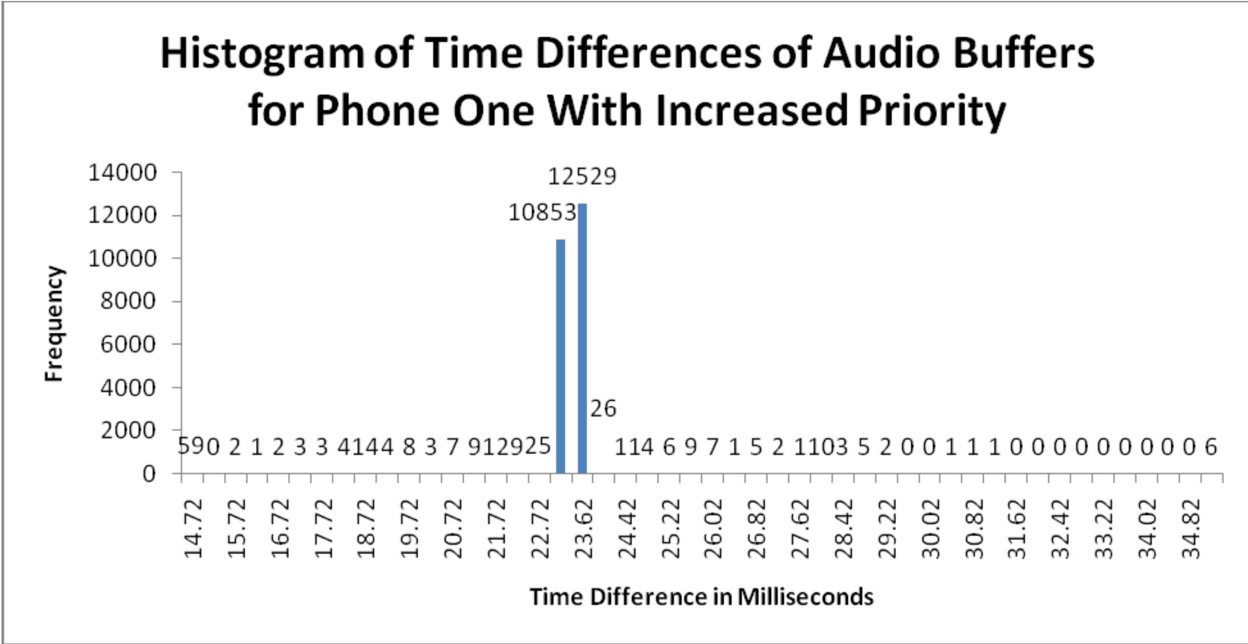


Figure 4 – (Top) Histogram of Differences in Timestamps Recorded Before Each Audio Buffer of Data for Phone One. (Bottom) Histogram of Differences in Timestamps Recorded Before Each Audio Buffer of Data for Phone Two. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

Although the time data differences center a small amount above 23.22 milliseconds for both phones, there are enough outliers to cause concern. Only 57.2 percent of the time differences for phone one and 52.67 percent of the time differences for phone two fall within one millisecond of

the expected 23.22 milliseconds. The method was improved by increasing the priority of the program using the Linux command nice with a priority of -20. The results of this test can be seen in Figure 5. Again, this test and the results shown are from the beginning of this project, but are comparable to the results of the ALMOND project.



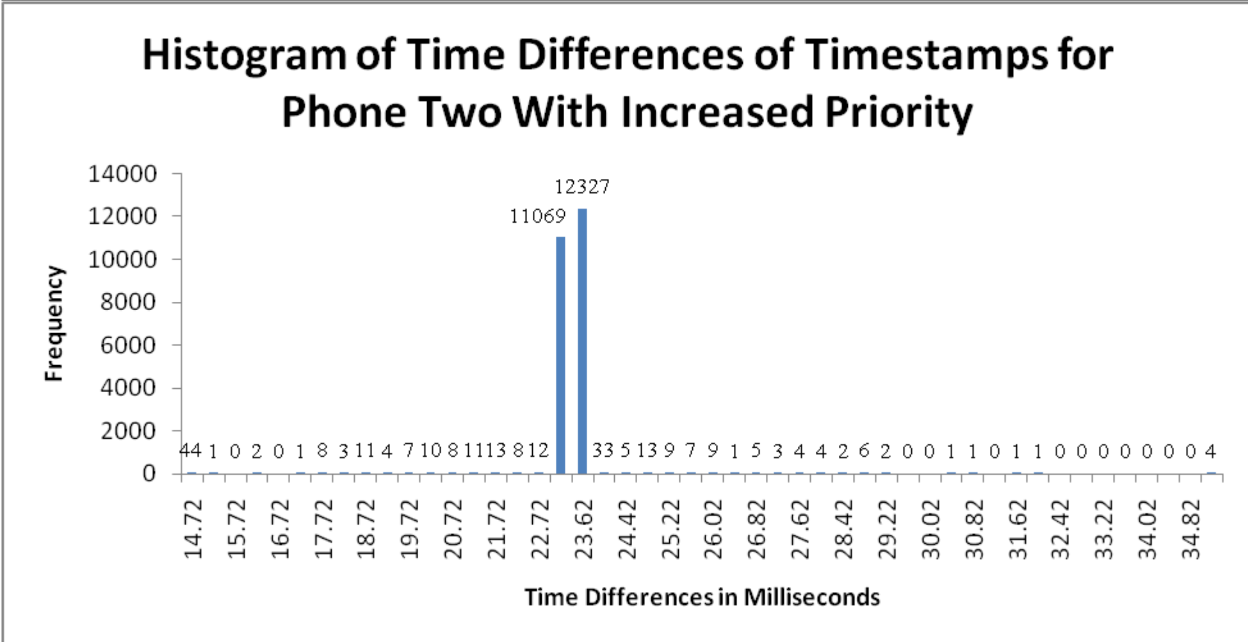


Figure 5 – (Top) Histogram of Differences in Timestamps Recorded Before Each Audio Buffer of Data for Phone One with Increased Priority. (Bottom) Histogram of Differences in Timestamps Recorded Before Each Audio Buffer of Data for Phone Two with Increased Priority. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

Increasing the priority of the program creates timestamps that are far more consistant with our expectations. 98.98 percent of the time differences for phone one and 98.97 percent of the

time differences for phone two fall within one millisecond of the expected 23.22 milliseconds. However, increasing the priority also has other undesirable effects. The main problem is that other programs will not be granted as much access to the CPU while the priority is so high. Furthermore, even though outliers have been reduced, they have not been eliminated, and one cannot completely rely on the time data without somehow correcting for them. Recall that in acoustic localization, even a few milliseconds of error can greatly reduce the ability to locate a sound source. For more information on the relationship of timing errors to positional errors, please refer to section 2.6.5 of [ALMOND, 2009].

2.1.2. Synchronizing the Time Between Two Phones

Originally, the ALMOND project sought to use the Global Positioning System (GPS) receiver on the Openmokos to synchronize the clocks between the phones. GPS would have given the phones the ability to synchronize to a time source that is accurate to within 100 nanoseconds 99 percent of the time [GPS datasheet, 2009]. Unfortunately, it was discovered that although the physical GPS chip on the phone was capable of delivering a signal with this precision and accuracy, the rest of the phone was not built to take advantage of this feature. Because the program to collect audio data had to run at such a high priority, the options available for synchronizing time between phones was limited. The most successful method for synchronizing the phones involved sending User Datagram Protocol (UDP) broadcast messages. Before starting to collect audio data, a UDP broadcast message was sent to both phones from the same computer which was playing chirps. When the phones received the broadcast message, they immediately reset their clocks to a known time. The message would be received at both phones at approximately the same time, which should have resulted in a high amount of precision of the clock times between the two phones.

To test the precision of times between the two phones, the ALMOND project physically positioned the phones as shown in Figure 6. The microphones of both Openmoko devices were positioned so that they were both equidistant from the center of the speaker. A sound played by the speaker should be heard by both phones at the same time.

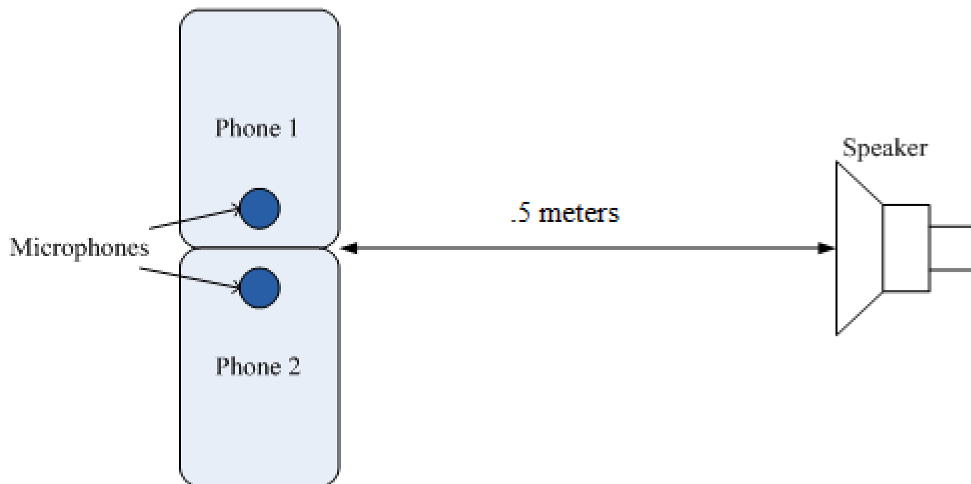


Figure 6 – Physical Setup to Test Time Synchronization Using a User Datagram Protocol Broadcast Message

We could test the synchronization between the two phones by playing sounds, and recording the time at which each phone detected it. If the phones were perfectly synchronized, then they both would detect the sound at the same time. Assuming the worst-case difference in the path length from the sound source to each microphone (due to errors in manual alignment/placement of the phones) of one centimeter, the maximum error (due to misalignment) in time arrival of the speaker-generated sound would be 29.4 microseconds. The resulting errors beyond this time can be attributed to errors in our methods of collecting time data from each phone, errors in detection, or differences in the times reported by each phone. A plot of the results of this test can be seen in Figure 7. Again, this test and the results shown are from the beginning of this project, but are comparable to the results of the ALMOND project.

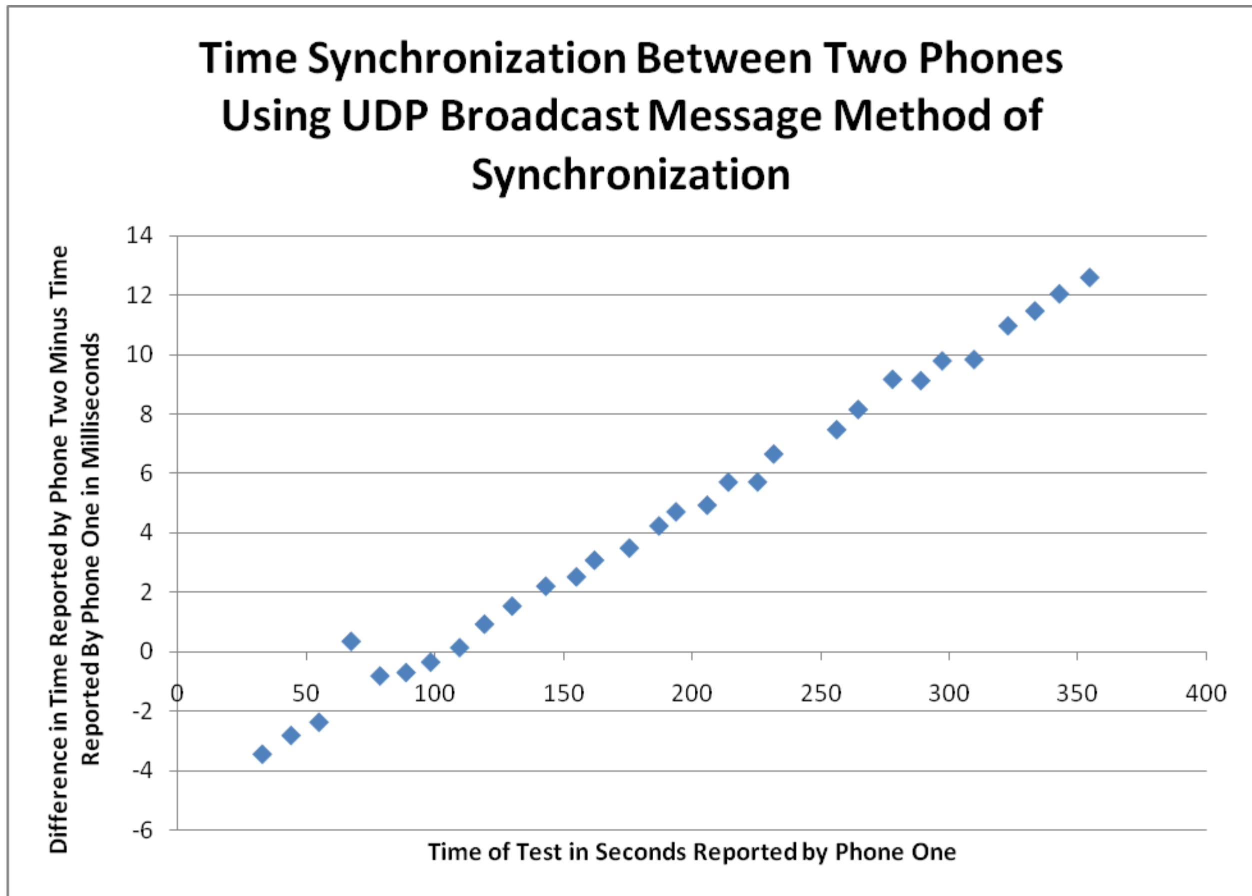


Figure 7 – Time Synchronization Between Two Phones Using an Initial UDP Broadcast Message Method of Synchronization. Each Point Represents an Audio Sample That was Detected.

The graph in Figure 7 reveals two challenges that must be overcome in order to synchronize the time between the two phones. The first challenge is the drift of the clocks away from each other over time at a rate of approximately 50 microseconds per second (potentially due to slight differences in the frequencies of the clocks on both phones). The second challenge is the initial synchronization. The drift appears to be fairly linearly, if a best fit line is drawn, the initial synchronization for this test would be -4.77 milliseconds. Due to time limitations, no extensive tests were done in the initial ALMOND project to attempt to characterize the initial synchronization values.

3. Obtaining More Precise Timestamps Through ALSA API

The tests that were run as part of the ALMOND project gave a starting point for this project. The two main goals were based on the problems faced:

1.) Increase the precision of the timestamps which an individual phone reports.

2.) Even if the precision cannot be substantially improved, use multiple individual timestamps to improve the overall precision.

This following sections will detail tests done for this project to address the first goal of improving the precision of retrieving timestamps.

Ideally, instead of retrieving the timestamps on the phones through the glibc `gettimeofday` function call, we would like to abstract the method for collecting time data and instead simply retrieve the timestamp associated with a particular sample when we retrieve the sample. That is to say, we would like to let the ALSA libraries associate timestamps with audio samples while it is sampling.

3.1. Confirm All Audio Data are Being Recorded

It was important to confirm that we were indeed receiving all of the audio data that we would expect. If, for some reason, we do not record every sample over a given period of time, then we cannot be sure if an incorrect timestamp is because of errors in retrieving the timestamp or if we simply missed audio data.

To test whether we were receiving all of the audio data stream, we connected our two Openmoko Freerunners to our central processing point through an 802.11g wireless network. The Openmoko devices were set up to run the same program used to measure the time in between reading each audio buffer as described in section 5 of the ALMOND project (Read and Queue Execution Time Test). Briefly, the phones would read and queue blocks of 1024 audio

data samples and stream these data over the network to the central processing point. Additionally, each phone would record the timestamps (one associated with each block) that they retrieved to a file on its disk. Both Freerunners contain hardware version A6 and run Debian GNU/Linux with version 2.6.29 of the kernel. The phones were set to sample audio data at a rate of 44,100 hertz.

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 audio and time data were processed on this computer in MATLAB version 7.8.0.347 (R2009a). For this experiment, MATLAB established a connection to the phones, received audio data, and reported the number of samples received over the duration of the test. The wireless network between the phones was set up using a Netgear FWG 114P wireless router with firmware version 2 using WPA2 certification.

The test was ended by a manual command entered to MATLAB and then relayed to the phones. The test was run for 549.04 seconds as measured independently by the difference between the first timestamp and last timestamp of both phones. During this period of time, MATLAB reported receiving 24215552 samples from one phone and 24217600 samples from the other. The difference between the two reported values is 2048 samples. The phones could have sampled different amounts for a variety of reasons. When MATLAB sends the command to start the test, it will send one phone the message before the other. Furthermore there could have been a slight difference in the rate at which the phones sample audio data. Also, the phones only send data when they have a full buffer to send, meaning the actual difference in samples could be different than reported. Finally, the phones will not send the audio data if the function call to

send the data would block; therefore, one phone may have not been able to send the last samples. The number of samples over the given period of times corresponds to sampling rates of 44,105 and 44,108 hertz respectively. These sampling frequencies are actually larger than the set sampling frequency of the test which was 44,100 hertz. This test showed that we were receiving more samples than we would expect. It should be noted that, while this test does not provide definitive proof that the correct number of samples is being provided, the results are consistent with our expectations of receiving the correct number of samples when factors such as clock differences and start up times are considered.

We also want to know that not only are we receiving the right amount of samples, but we are receiving the correct samples. To test the sample's correctness, we modified the program on the central processing point to record the audio samples that were being sent. We then used the central processing point to play a 500 hertz sine wave through a single Altec Lansing Computer Speaker ACS90. The speaker was set at a volume such that the sound at the microphone was 82 dB. The ambient noise of the room was 56 dB. After the test completed, a manual inspection of approximately ten seconds of the data was conducted. A sample of the recorded audio data can

be seen in Figure 8.

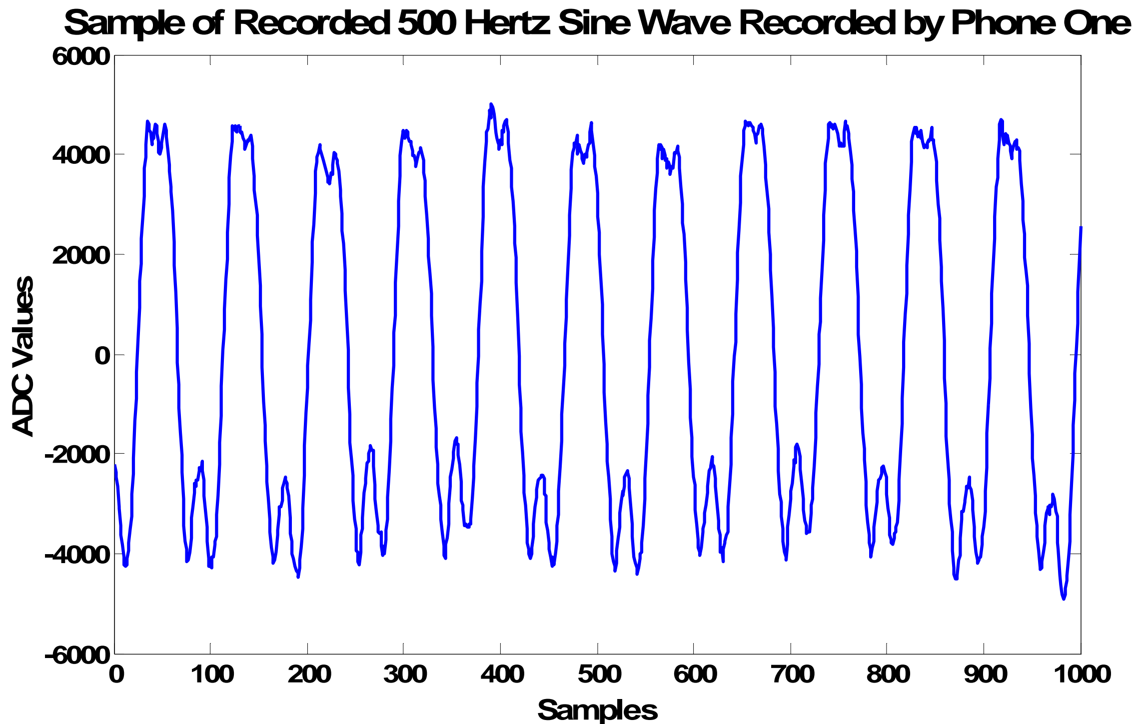


Figure 8 - A Sample of Recorded Audio Data when Playing a 500 Hertz Sine Wave. The Recorded Data Shows No Obvious Discontinuities or Phase Shifts Except at the Peaks of the Sine Waves.

The recorded audio data seems to fluctuate near the peaks of each period of the sine wave. These fluctuations could be due to the microphones not providing a steady voltage for the analog to digital converter. Still, the periods for each sine wave are consistent. During the manual inspection there were no noticeable discontinuities (other than near the peaks of each period, as described above), and there were no noticeable phase shifts. It should be said that this test does not prove that no audio data are being lost. More tests will have to be done to show that every sample is captured which cannot be performed here due to time constraints. The results indicate that it is not unreasonable to assume that no audio samples are being lost and audio is being sampled at a constant rate. For the rest of this paper, it will be assumed that no audio samples are being lost.

3.2. Obtaining Timestamps From ALSA

The Advanced Linux Sound Architecture (ALSA) Application Programming Interface (API) provides functions to generate timestamps in between every audio buffer. By using the ALSA libraries to collect timestamps, we were able to lower the priority of the program, and with some manipulation, increase the precision of the obtained timestamps. Unfortunately, it is not clear whether the timestamps ALSA provides correspond to the end of one audio buffer or the start of the next. For this project, this distinction is not important since we are most concerned with comparing the time between timestamps (which would be consistent in either case).

3.2.1. Methodology

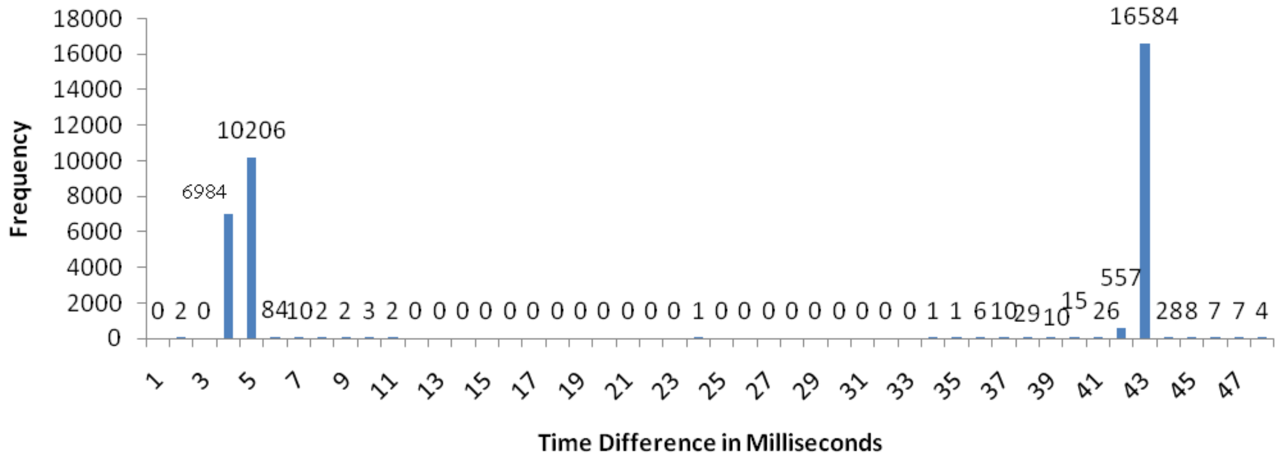
The method for obtaining timestamps starts with the program described in section five of the ALMOND project (Read and Queue Execution Time Test). It was modified to collect the timestamps from ALSA. The first step to modify the program was to allocate a structure for software parameters using the `snd_pcm_sw_params_malloc` function. The timestamp mode can then be set using the `snd_pcm_sw_params_set_tstamp_mode` and passing it a value of `SND_PCM_TSTAMP_MMAP`. The software parameters can then be enabled by using the `snd_pcm_sw_params` function. It is important to note that the `snd_pcm_sw_params` function must be called after all of the hardware parameters (sampling rate, buffer size, et cetera) are set or ALSA will not enable timestamps, although this sequence requirement is never clearly stated in the ALSA API. After timestamps have been enabled, one can query the timestamps simply by calling the `snd_pcm_htimestamp` function. This call will return the audio sample associated with the boundary of the last audio buffer read. The complete, modified program is presented in Appendix A. The program needs to be compiled using the default libraries provided by the Debian Openmoko operating system rather than the libraries provided by the toolchain on

Openmoko's Wiki. To satisfy the library requirements one must copy the `libasound` libraries from `/lib/` folder on the Openmoko into the toolchain's library directory on the compiling machine. [Kysela et al., 2009]

The program ran with high priority (using the Linux nice command and the highest priority of -20) and recorded the timestamps to a file located on the phone. Audio and these timestamp data were sent to the central processing computer, but ignored for this test since we are investigating the consistency of the timestamps on the phones.

3.2.2. Results and Discussion

We must assess the consistency of time interval between the timestamps (with the size of the buffer being 1024 samples the time would ideally be 23.22 milliseconds). We took the $(i + 1)^{th}$ timestamp and subtracted the ith timestamp to create a histogram of consistency for each phone. The histograms can be seen in Figure 9.



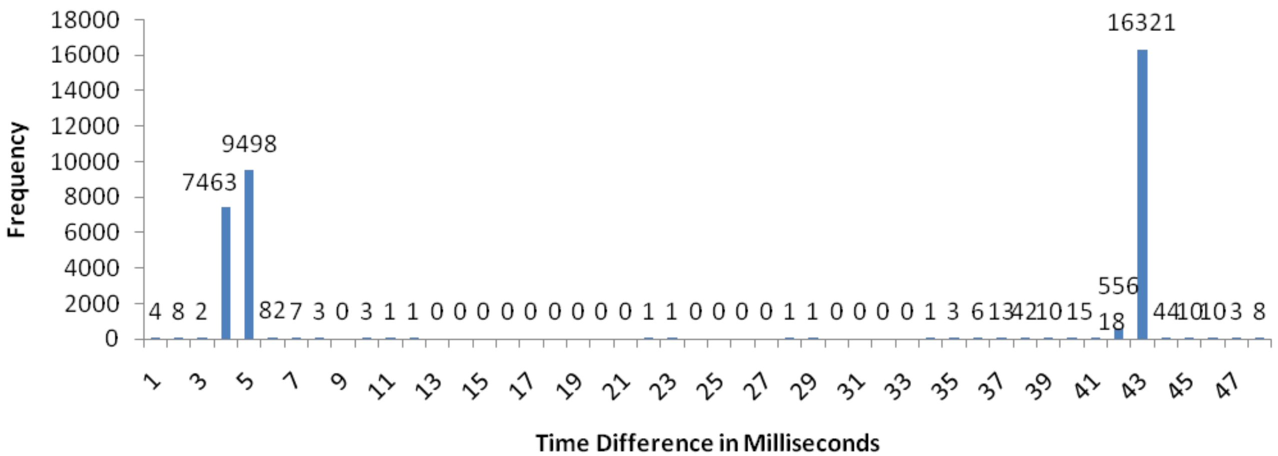


Figure 9 – Histograms Showing the Time Difference Repored by ALSA Between Audio Buffers in Phone One (Top) and Phone Two (Bottom). The Expected Value for the Audio Buffer Size and Sampling Rate is 23.22 milliseconds. The First Bin (One Millisecond) Corresponds to All Values Less than or Equal to One Millisecond while the Last Bin Corresponds to All Values Greater than or Equal to 47 Milliseconds.

Surprisingly, far less than one percent of the time differences are within one millisecond of the expected time difference. It is not clear why these groupings occur. However, the data seem

to be clustering around forty three milliseconds and four milliseconds, which, when averaged, is 23.5 milliseconds, close to our expected 23.22 milliseconds.

In Figure 10, we show an x-y scatter chart of the time differences reported between audio buffers. For each point, the x value represents a time difference, while the y value represents the time difference which immediately followed. We apply this process to the first 250 time differences to show a specific behavioral pattern.

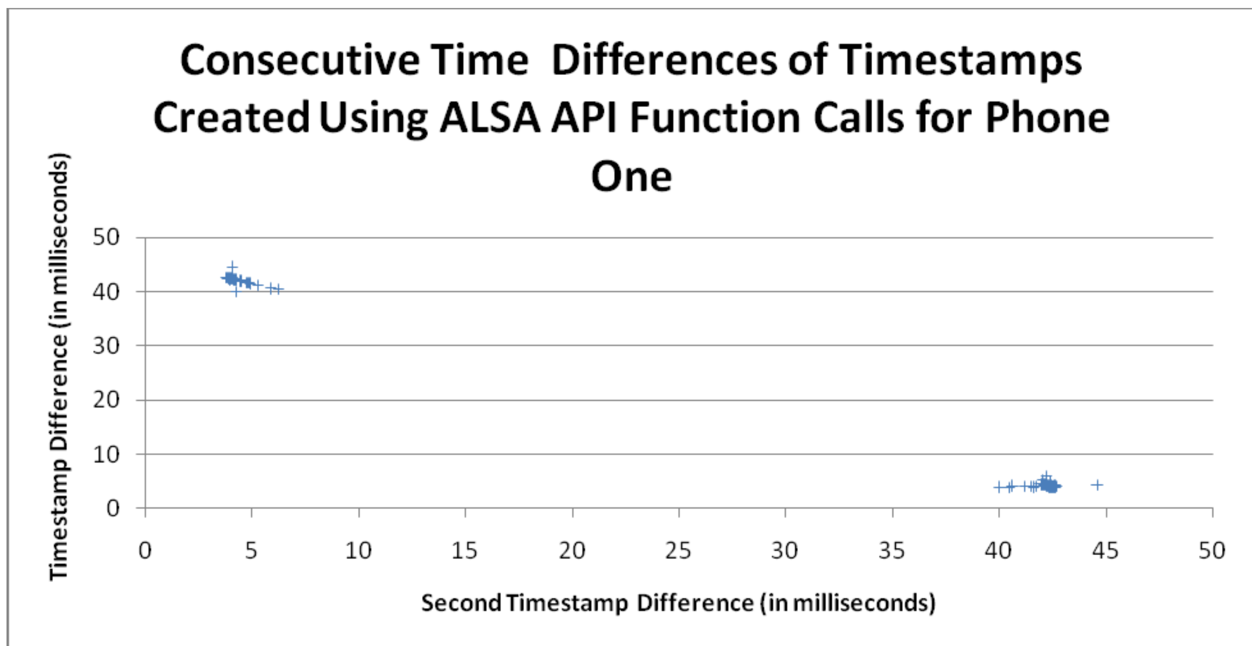


Figure 10 – Comparing Consecutive Timestamp Differences. The X-Axis Represents a Timestamp Difference, While the Y-Axis Represents the Timestamp Difference which Immediately Followed.

Figure 10 shows two distinct groups to the timestamps. In the upper left of the graph in Figure 10, there are timestamp differences which are approximately forty to forty-five milliseconds which were immediately preceded by timestamps of duration four to six milliseconds. The bottom right shows timestamp differences which are four to six milliseconds and were immediately preceded by timestamp differences which were forty to forty-five milliseconds. This graph suggests that the timestamp differences are alternating between the two ranges (i.e. four, forty two, four, forty two, and so on).

Figure 11 shows the timestamp difference histogram for each phone when each difference is averaged with its previous difference (a moving average of two).

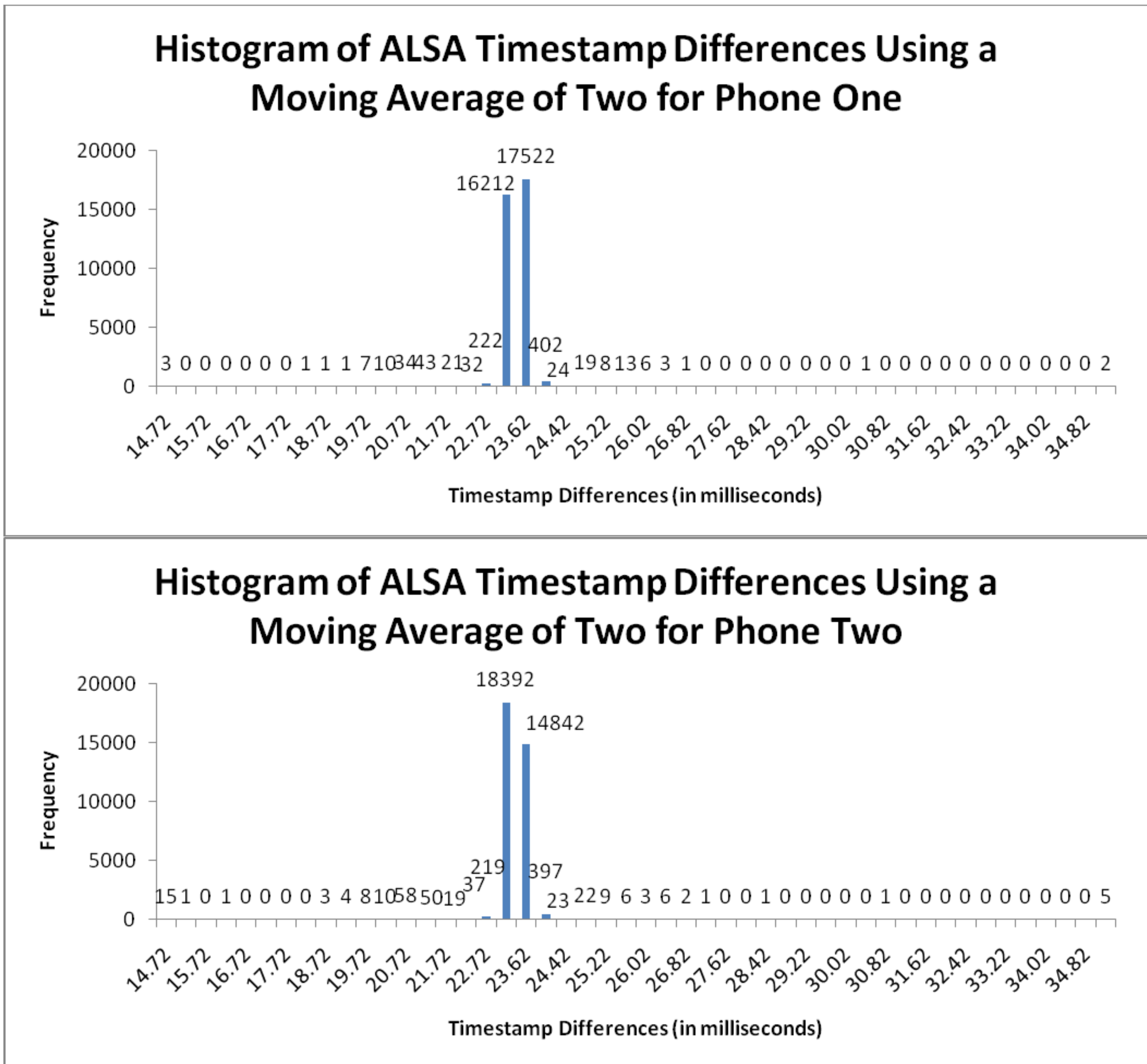
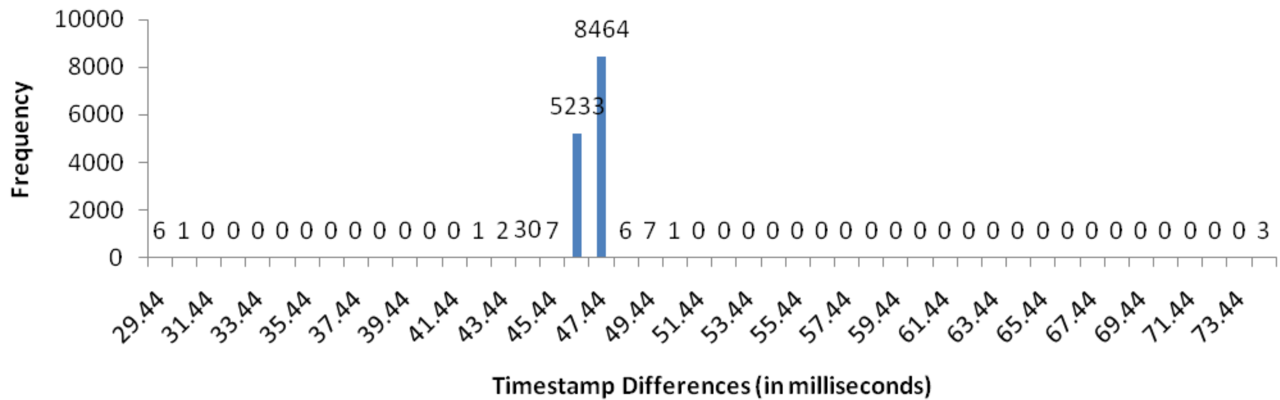


Figure 11 – Histogram of Timestamp Differences Using ALSA Timestamps and a Moving Average of Two. The Starting Transient (First Time Difference) was Discarded for Each Phone. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

These results are encouraging. There are fewer outliers than in the previous ALMOND tests and the extent of the outliers is less as well. When creating timestamps using the `gettimeofday` function, the maximum difference between two timestamps could be well over 400 milliseconds. With this test — retrieving timestamps from the ALSA API — the maximum time differences were 57.5 milliseconds for phone one and 125.6 milliseconds for the other.

It was still not known why the timestamps from ALSA seem to alternate, and if they alternate in any consistent way. To find out, the size of the audio buffer was changed from 1024 to 2048 samples and the test was rerun. With no running averages, the result from the test illustrated a bimodal distribution similar to the ALSA test with a buffer of 1024 audio samples, except that the modes were centered around eight milliseconds and eighty milliseconds. Figure 12 shows the resulting histograms when using a moving average of two.



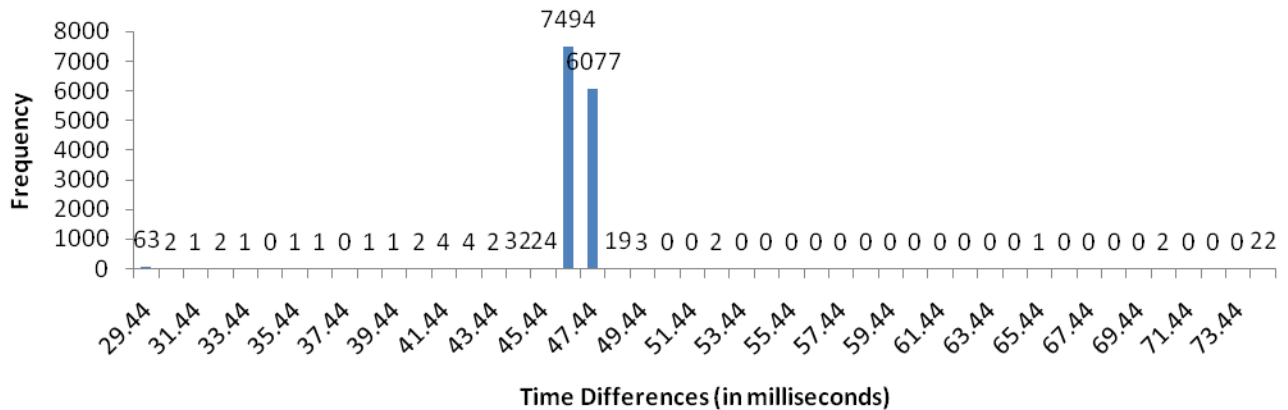


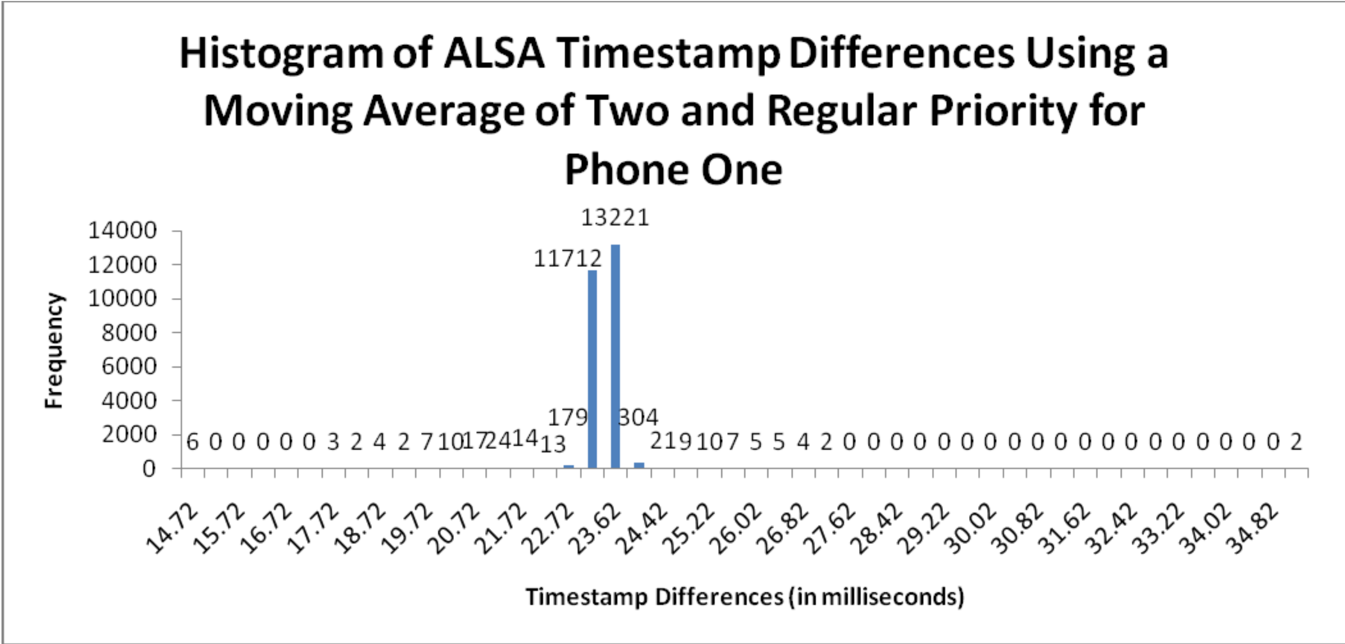
Figure 12 – Histograms of Time Differences Provided by ALSA Timestamps Using a Moving Average of Two and an Audio Buffer of 2048 Samples in Order to Display Consistency in Timestamp Reports. The Starting Transient (First Time Difference) was Discarded for Each Phone. The First Bin (29.44 Milliseconds) Corresponds to All Values Less than or Equal to 29.44 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 73.44 Milliseconds.

The results from the moving average with a buffer size of 2048 samples is expected. For 2048 samples at a 44,100 hertz sampling rate, the time difference between timestamps should be

46.44 milliseconds which is where we see the vast majority of the averaged differences clustered. Figure 12 suggests that, despite the fact there is an unknown factor which creates bimodal time differences when reading ALSA timestamps, a unimodal distribution can be made by producing a moving average of two time difference values. Furthermore, this unimodal distribution is more consistent than the approach taken in the ALMOND project (using using `gettimeofday` to retrieve timestamps).

3.3. Reducing the Priority of the Phone's Program when Using ALSA Timestamps

One of the potential benefits of using ALSA is the ability to reduce the process priority of the program on the phones. The API collects the timestamps after each buffer, so as long as the program and network can keep up with streaming and processing the data, our program should not need to be run in high priority (unlike previous tests in this section). To illustrate this, the program to collect ALSA timestamps was used with the buffer size of to 1024 samples and the program was run without any change in priority (without the Linux nice command). Figure 13 shows the resulting histograms of time differences.



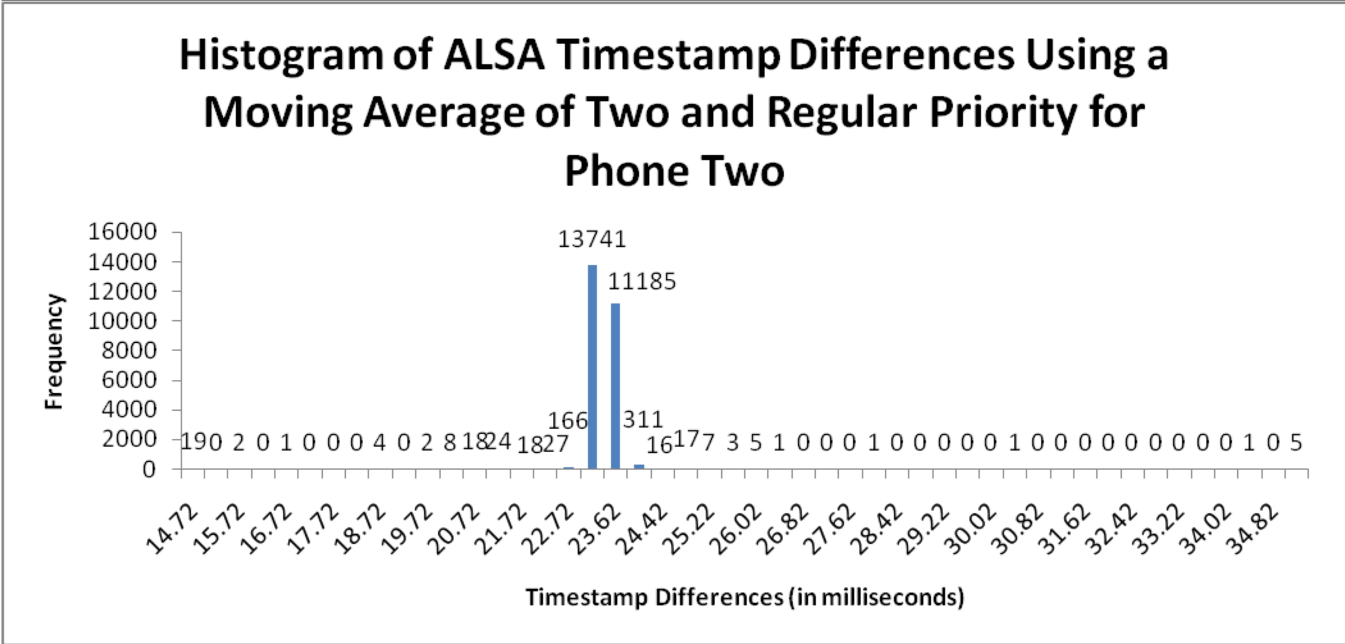


Figure 13 – Histograms of Time Differences Between Audio Buffers Using Normal Priority and a Moving Average of Two Time Differences. The Starting Transient (First Time Difference) was Discarded for Each Phone. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

Although outliers still exist, the vast majority of cases are still centered at 23 milliseconds. The maximum time differences have also not increased (92.70 milliseconds for phone one and 123.01 milliseconds for phone two). Thus, a higher priority does not seem to alter these results.

With these tests, we have achieved the first goal of this project: to improve the precision of the timestamps. Nearly 99 percent of timestamp differences are within one millisecond of the expected values. These tests have also demonstrated the useful benefit of being able to lower the priority of the program to read audio data and timestamps. In the future, lower priority makes it easier to incorporate other programs into the design of a system by allowing more access to the CPU. However, approximately one percent of timestamp differences differ from the expected difference by more than one millisecond. The maximum time difference is nearly one hundred milliseconds greater than the expected value. We cannot rely on these outliers for acoustic localization and must, therefore, investigate ways to filter out outlier timestamp differences.

4. Modifying Timestamps to Improve Precision

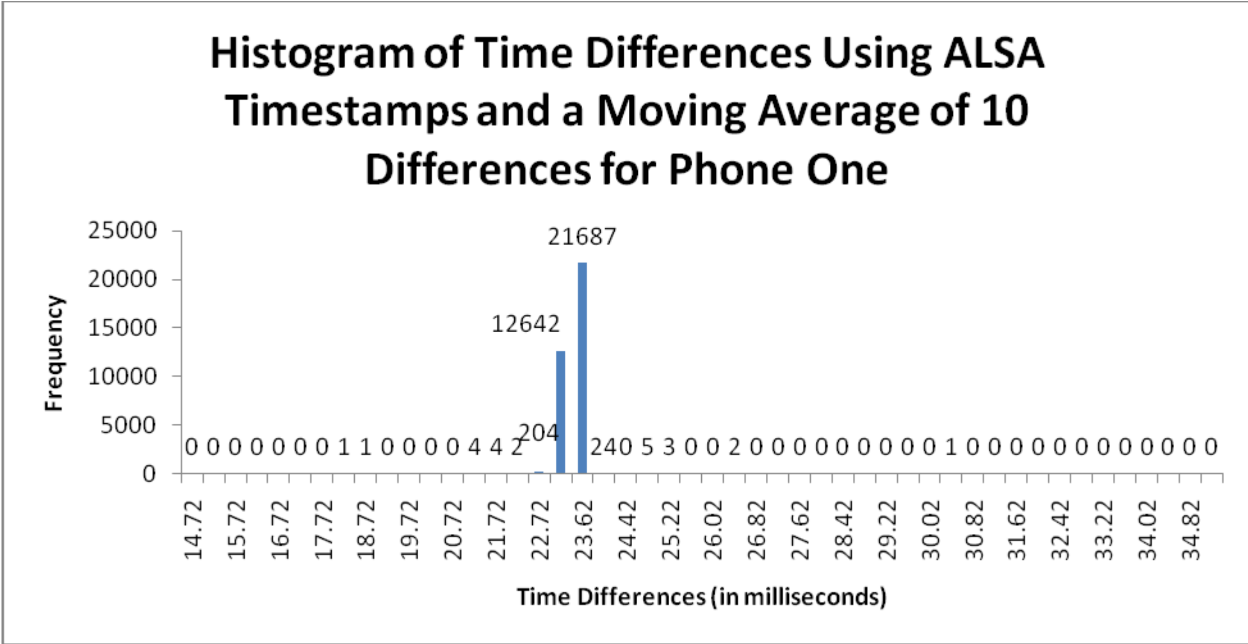
In the previous section it was determined that although the accuracy of the timestamps collected can be improved, there are still a significant number of timestamp differences which are outliers. Recall that these outliers are problematic for source localization. Given the hardware and software platform, it would not be reasonable to assume that one would ever achieve “perfect” timestamps. However, using methods to correct for errors in timestamps may result in more precision than is possible by simply using individual timestamps.

4.1. Moving Averages

Perhaps the simplest model that comes to mind when considering correcting for erroneous timestamps is to perform a moving average of the differences between timestamps. If individual timestamps include a random error component, then averaging would reduce this error. We have already seen that the ALSA timestamps depend on a moving average of at least two values, but what happens when more timestamps are included in calculating the result?

For this question, we will use the same time data from the previous test using ALSA timestamps, a 1024 sample buffer, and normal priority. These settings are the best choice for our analysis because they could be used in a real world application and because we can easily compare new results to previous results.

Figure 14 shows a histogram of the moving average of ten timestamps after discarding the first nine transient values for the test. Note that the moving average must incorporate an even valued number of timestamp differences due to the bimodal distribution of the timestamps.



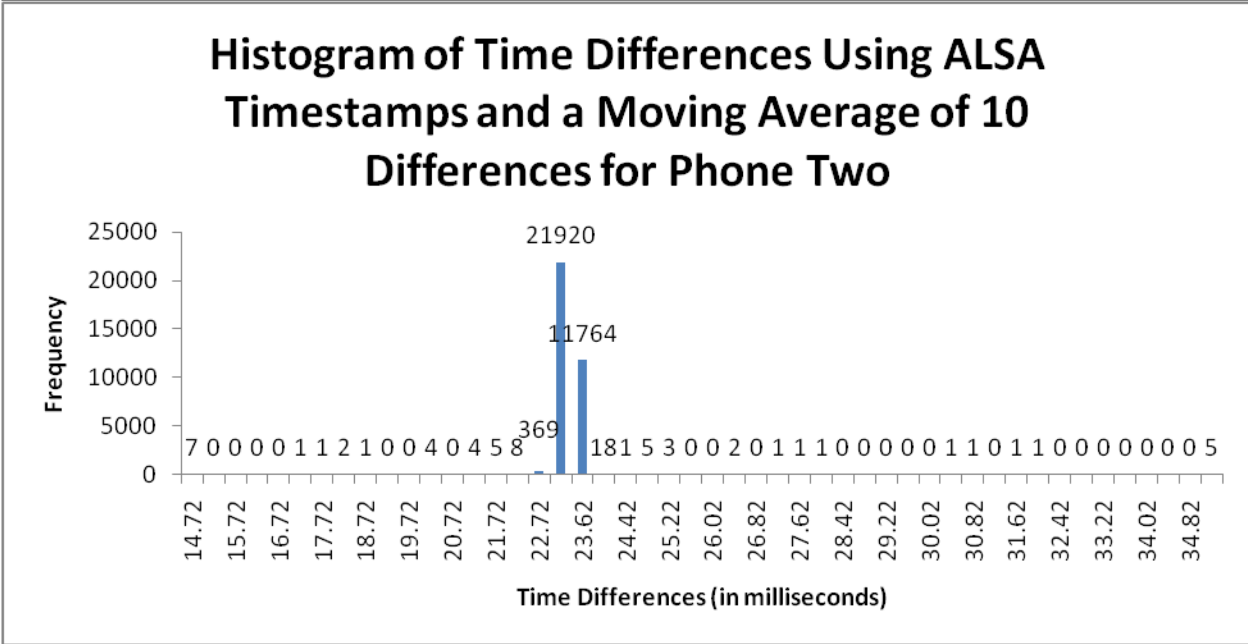


Figure 14 – Using a Moving Average of Ten Differences Between ALSA Timestamps to Improve Consistency and Precision The Starting Transient (First Nine Time Differences) were Discarded for Each Phone. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

The results of the moving average are not completely expected. While the maximum differences between timestamps has been successfully reduced in both cases (30.10 milliseconds for phone one and 43.68 milliseconds for phone two), for phone two there has been no reduction

in the number of time differences greater than thirty-five milliseconds. This phenomenon occurs because there are a few time stamps which create such great outliers that they cannot be easily corrected for with averages of ten values. However, timestamps are acquired at a rapid rate. If one were to use 100 timestamp differences in a moving average, for example, it would take less than 2.4 seconds before the system could be operational. With higher averages, any individual timestamp difference's effect can be reduced. To demonstrate this, a moving average of 100 was performed on phone two's timestamp differences using ALSA timestamps. The results are shown in a histogram in Figure 15. Phone two was used because with a moving average of 10 there were more outliers of greater magnitude than phone one.

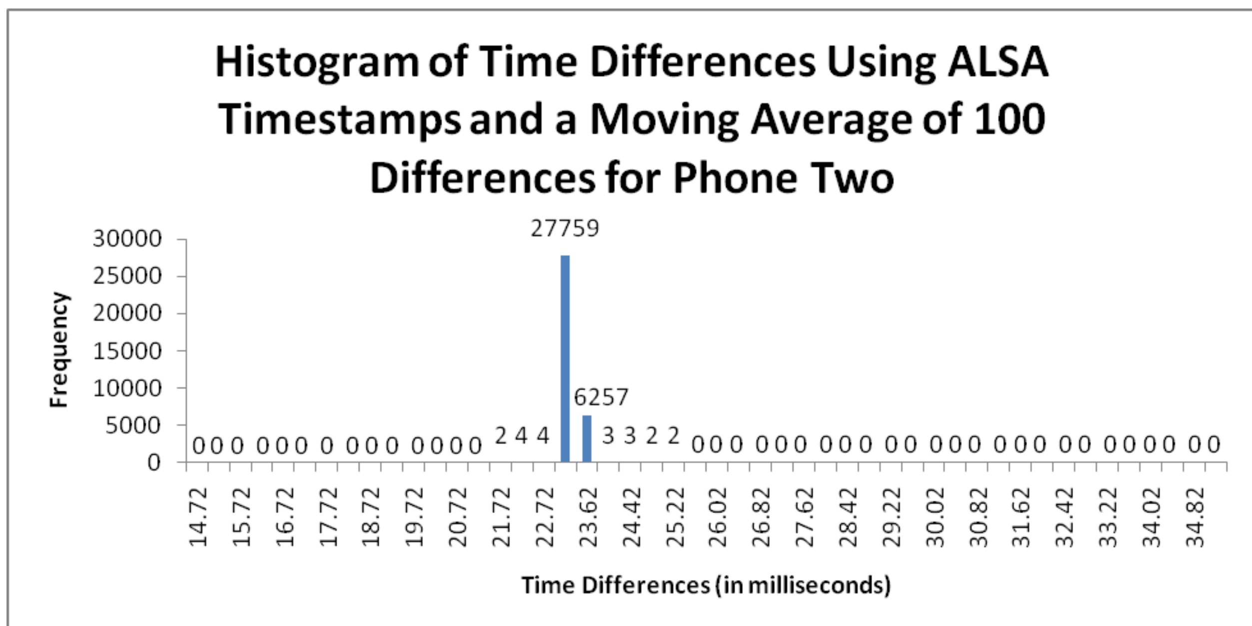
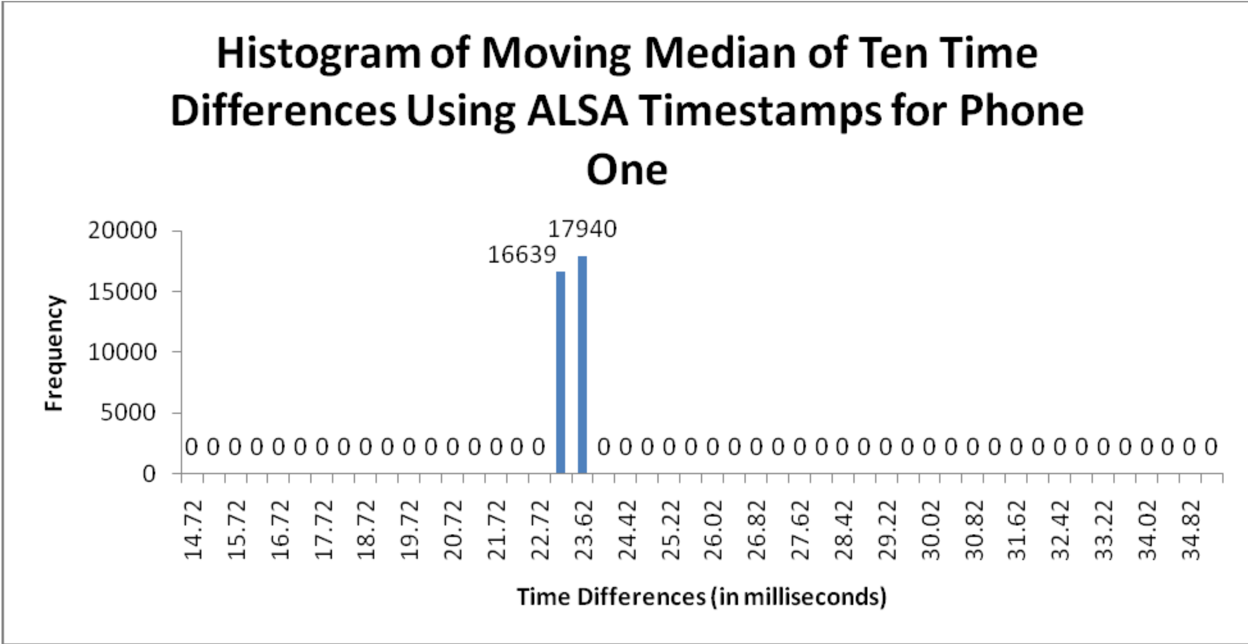


Figure 15 - Using a Moving Average of 100 Differences Between ALSA Timestamps to Improve Consistency and Precision The Starting Transient (First 99 Time Differences) were Discarded. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

Using a moving average of 100 on phone two's time data from ALSA timestamps, all time differences are now within 3 milliseconds of the expected time difference, while 99.9 percent are within one millisecond. This is a large step forward from the ALMOND project.

4.2. Moving Medians

Instead of using a moving average approach, it may be beneficial to consider taking advantage of the fact that the vast majority of time differences are the values we wish to have. To do this, we first start with a moving average of two because without the moving average of two, it would be likely that the median would fall somewhere around four milliseconds or forty milliseconds due to the bimodal distribution of the ALSA timestamp differences. We then do a moving median of ten values of the moving average. The resulting histogram of the time differences is shown in Figure 16.



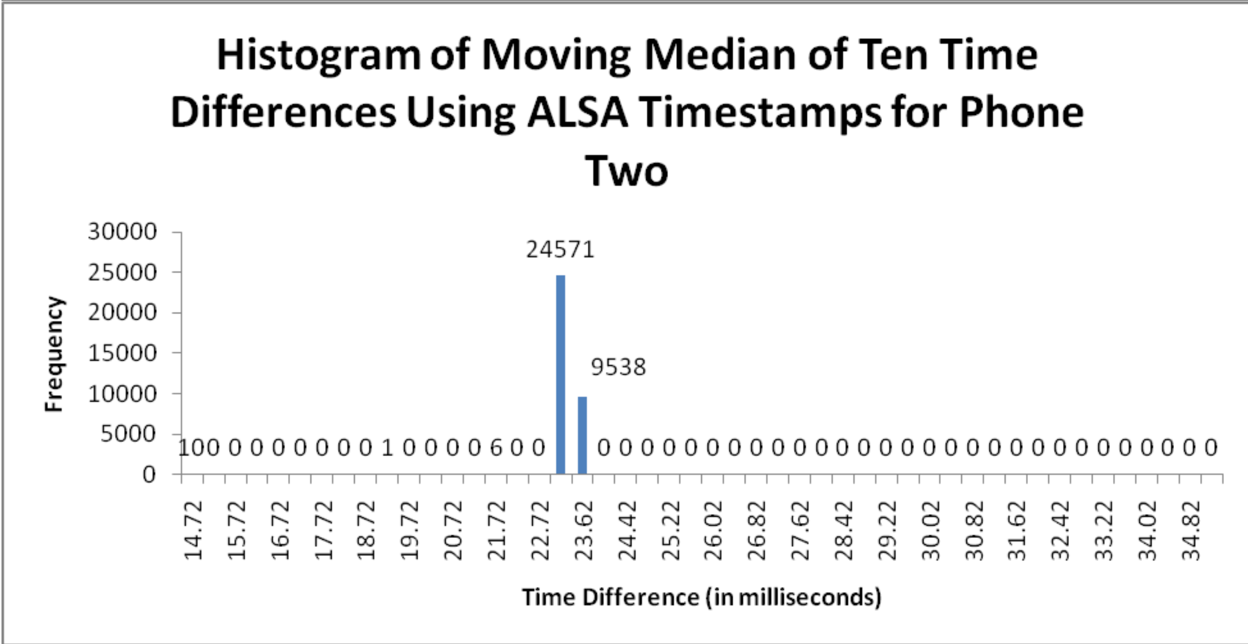


Figure 16 – Histograms of a Ten Sample Moving Median Applied To Time Differences Created from Applying a Moving Average of Two to ALSA Timestamps. The Starting Transient (First Nine Time Differences) were Discarded for Each Phone. The First Bin (14.72 Milliseconds) Corresponds to All Values Less than or Equal to 14.72 Milliseconds while the Last Bin Corresponds to All Values Greater than or Equal to 34.82 Milliseconds.

These results show a huge improvement in consistency over the last results. In all of the data collected there are only eight values which were not within a millisecond of 23.22 milliseconds expected. The test discards the first nine values (since this is a moving median of ten values, the

transient values should not be included). This indicates that the outliers are a long string of timestamp differences that occur. It is unknown what causes these strings.

It may, however, not be prudent to use moving medians in application. Although the vast majority of timestamp differences fall within a millisecond of the expected values, small errors in individual differences may accumulate over a long period of time to produce undesired drift. More work will have to be done to know whether moving medians can be effectively utilized.

4.3. Synchronizing Time Between Phones

Using moving medians and moving averages along with timestamps retrieved through the ALSA API, we can retrieve the time differences in a consistent and precise manner, we need to examine the second issue of time synchronization: the ability of the phones to synchronize their clock times with each other. The data collected during the ALMOND project seemed to indicate that the phones would drift away from one another at a linear rate of approximately fifty microseconds per second. Due to time constraints, it was unclear how closely they could be synchronized initially. For this project tests were performed to measure the extent of this initial synchronization.

4.4. Methodology

The hardware setup for this experiment is exactly the same as from the previous experiments to test UDP synchronization (see section 7 of [ALMOND, 2009]). Briefly, the two Openmoko devices are placed such that their microphones are equidistant from a speaker and the line passing through the microphones was perpendicular to the midpoint of the speaker 0.5 meters away (see Figure 6). The distance was measured using a tape measure which has labels for centimeters. Assuming placement errors of less than one centimeter the maximum time difference of arrival caused by errors in measurement will be less than thirty microseconds. The

speaker, Altec Lansing Computer Speaker System ACS90, was set to a volume such that a five kilohertz sine wave played at a distance of one half meter produced 87 dB Sound Pressure Level (SPL) as measured by Radio Shack Digital-Display Sound-Level Meter (model 33-2055). The ambient noise of the room was measured to be 57 dB SPL.

The program used on the phones was the same program as described in section seven of the ALMOND project (Time Synchronization Using User Datagram Protocol Method). This program was used due to convenience. Briefly, the phones would stream sound and audio data over the network to the central processing point. At the central processing point, the sound data would be analyzed looking for a 100 millisecond chirp which varied in frequency from 100 to 20,000 hertz. The program was run at the highest priority using the Linux nice command. All sounds and the UDP broadcast messages were sent from a Dell desktop running Fedora 10 Linux with kernel version 2.26.27 with a 2.8 gigahertz Pentium IV processor and 512 megabytes of RAM.

At the beginning of the test, the phones would start listening for a UDP broadcast message from the Fedora computer. When the phones received the message, they would reset their clocks to a known time. Thereafter, the phones would start to record and send audio and time data to the MATLAB computer. A chirp sound was then manually initiated, and the time of occurrence was noted for each phone. All of these steps took place within ten seconds of each other. Given the previously discovered drift rate of approximately 50 microseconds per second, the values recorded could have a bias of up to .5 milliseconds, a small enough error for a rudimentary comparison. This bias could be reduced in the future by creating an automated method of performing the test. The test was stopped after the first chirp and the entire procedure (listening

and playing a UDP broadcast message, resetting of the phone's clocks, recording audio data, playing a chirp, and noting the time difference) was repeated 50 times.

4.5. Results and Discussion

Under ideal conditions, the difference of arrival times of the chirp at each of the phones should be zero. The UDP broadcast would get to each phone at the same time, both phones would take the same amount of time to reset their clocks, there would be no drift, and the time values would be read correctly for all audio.

Figure 17 shows the results of the time differences between detection of the chirp between the two phones for this UDP test when run fifty times.

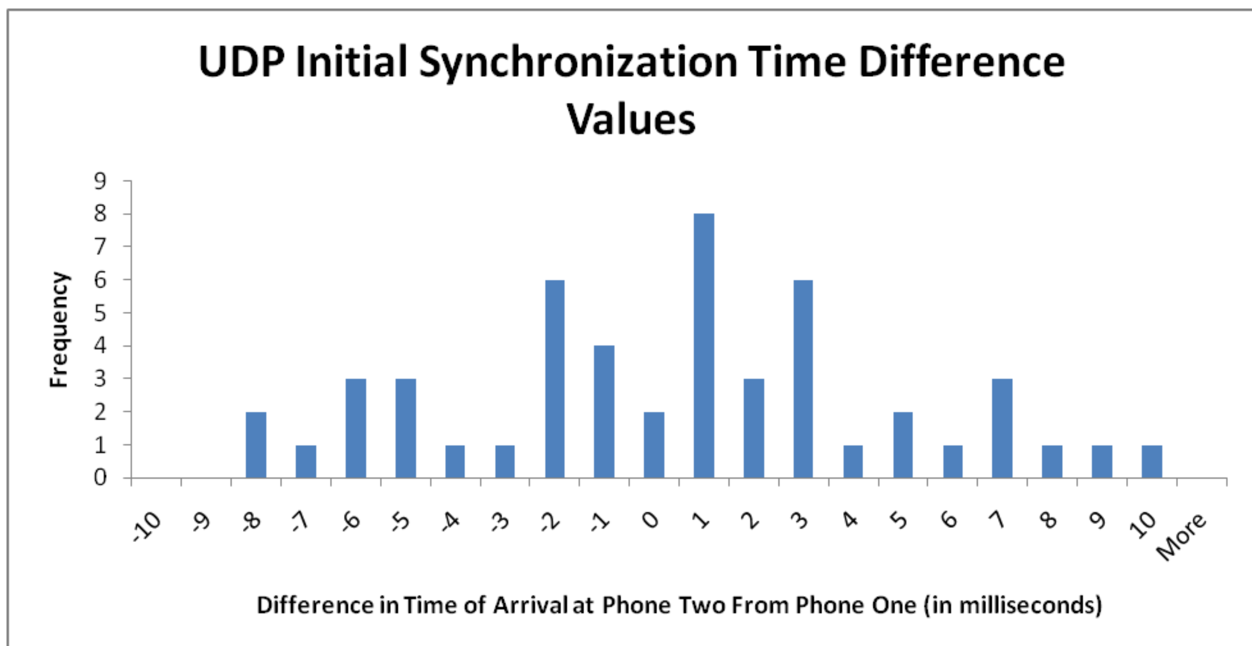


Figure 17 – Initial Differences in Time Synchronization when using a UDP Broadcast Message for Synchronization

These results indicate that it would indeed be possible to get a high amount of precision when synchronizing phones using UDP broadcast messages, however, combined information from several messages sent for synchronization would be required. The average value for the initial synchronization was -0.0733 milliseconds (meaning phone one was .0733 milliseconds behind

phone two) with a standard deviation of 4.38 milliseconds. The average value is in opposition to the drift, as in the test to synchronize the phones using a UDP broadcast signal (see section seven of [ALMOND 2009]) it was found that phone two's clock would run faster than phone one. These results seem to indicate that the initial synchronization is approximately zero centered, so simple averaging of the times should suffice to improve synchronization.

5. Discussion

This project was successful in achieving its two main goals. Firstly, we were able to retrieve more precise timestamps at a lower priority by using the ALSA API. Secondly, we were able to take the timestamps and manipulate them with moving averages and moving medians to be able to achieve even more precise results.

There are still several problems facing acoustic localization using phones, however. One of most pressing issues is to determine if ALSA is recording all the necessary audio data. Our intial tests in section 3.1 indicate that it is not unreasonable to assume that all data is being recorded, but it has not been proven. One way of performing this test would be to remove the microphone from an Openmoko and replace it with a function generator to generate a consistent sine wave. It would then be easier to search recorded audio data for discontinuities or phase shifts.

To date, no method has been implemented to continuously synchronize the time between two phones. In section four of this project, it was postulated that UDP broadcast messages could be used for this purpose. In such a system, the central processing point would send out periodic UDP messages, when the phones received the messages, they would retrieve a timestamp and then send the time data back to the central processing point. The central processing point would determine how to correct the offset of the phones while accounting for the drift. If this method were implemented, it may even be possible to eliminate ALSA timestamps. If the central processing point can determine the initial time offset and can account for the drift of the phone's clocks, then theoretically the time can be determined based on the number of audio samples received. Of course, this is only one possible method of using UDP broadcast messages. One could postulate others. The important point to take away is that with the average initial synchronization achieved by UDP messages described in section 4.3 being close to zero (.0733

milliseconds off), it should be possible to synchronize between the phones (using averaging) to within a millisecond of error.

Other possible methods of achieving time synchronization between phones were discussed in [ALMOND, 2009]. Briefly, one could use GPS to synchronize the clocks of the phones. GPS receivers are able to determine the time precise to within a few hundred nanoseconds, which should be more than enough for acoustic localization. Unfortunately, GPS is not available in all locations where acoustic localization may be useful, for example, indoors or in urban areas with many tall buildings. In addition, this work has shown that even if precise time is available on a phone, time can not necessarily be associated with audio data samples with the same amount of precision. Another method would be to use NTP, which works by having the phones query a reference time base using UDP messages. From many of these messages, the phone will slowly correct its clock drift until it is synchronized with the time base.

6. Conclusions

The first goal of this project was to obtain more precise timestamps from the hardware. The first step towards achieving this goal was fetching timestamps from the ALSA API. Using this method outright, resulted in less than one percent of differences between successive timestamps being within one millisecond of the expected values. However, it was discovered that the resulting ALSA timestamps had a bimodal distribution, with timestamp differences alternating between the two nodes. Using this information, a moving average of two was applied to the timestamp differences resulting in over 99 percent of the timestamp differences falling within one millisecond of the expected values. Using ALSA timestamps provided the added benefit of being able to reduce the priority to normal levels and still have over 98 percent of the timestamp differences fall within one millisecond of the expected values.

The second goal of this project was related to reducing the number of outliers present even when using ALSA timestamps. To reduce the number of outlying timestamp differences, the differences were filtered using both moving averages and moving medians. With a moving average of 100, all timestamp differences fell within three milliseconds of the expected value, while 99 percent were within one millisecond of the expected value. A moving median of ten timestamp differences provided 99 percent of values within one millisecond, however it is still unclear whether a median can be applied in practice due to the potential for small errors in individual differences accumulating over time causing drift. Moving medians also revealed long strings of timestamp differences that were much shorter than expected (over ten milliseconds less than the expected values). The causes of these short timestamp differences is still unexplained.

This project also further characterized the errors in synchronization between phones by investigating the initial synchronization using a UDP broadcast message. The average initial

synchronization between two phones had a mean of 0.0733 milliseconds. This indicates that one could use multiple UDP messages in order to synchronize the phones to less than a millisecond, even given the bias of up to .5 milliseconds of the test.

Given the ability of UDP broadcast messages to synchronize the time between the phones to less than a millisecond, and the ability to retrieve timestamps from a phone with precisions of less than three milliseconds, it should be possible to detect when audio samples occurred precise to within four milliseconds. Of course, four milliseconds would be a worst case scenario. We have shown that over 99 percent of timestamp differences are within one millisecond, and would therefore conservatively expect to be able to determine the time at which an audio sample was taken precise to within 2 millisecond the vast majority of the time. These results are far better than the initial ALMOND project which could achieve synchronization to within ten milliseconds but only for two to three minutes at a time.

Although this project and the ALMOND project made significant progress towards acoustic localization using cellular phones, there are still a number of hurdles that must be overcome before a viable system can be implemented. Although using moving averages and moving medians has been shown to increase the performance of the system, no method for actually performing these calculations in real time has been implemented. Furthermore, a method will need to be implemented to synchronize the clocks between the phones, taking into account the differences in initial synchronization by sending out multiple UPD broadcast messages and taking into account the drift rate which is relatively constant between the two phones. Time synchronization could be custom built, or (since the priority on the main program was reduced) one may take advantage of an already implemented method of synchronization such as NTP. An

even better option for future phones if the hardware supports it would be to directly use GPS as a time base for synchronization.

These two projects have shown some of the limitations and many of the capabilities of cell phones as a platform for development. Although there are many challenges ahead, the viability of using cellular phones as mobile sensors is apparent.

7. References

Almquist, S., Skehan, D., & Saleem, M. (2009). Acoustic localization for mobile open-source network deployment (ALMOND). Major Qualifying Project Report, Worcester Polytechnic Institute.

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

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.

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

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

Kysela, J., Bagnara, A., Iwai, T. & Pol, F. v. d. (2009). *PCM (digital audio) interface.* Retrieved December 3, 2009, from http://www.alsa-project.org/alsa-doc/alsa-lib/pcm.html

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

Appendix A: Program to Retrieve and Record Timestamps using ALSA API Function Calls

```
/* 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>
#include <alsa/pcm.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[]) {
  FILE *file;
  file=fopen("gettimes.txt", "a+");

  /***************internet******************/
     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;
     }
      //This is the cromulent way to do TCP/IP internet
     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;
      snd_pcm_sw_params_t *swparams;

      unsigned int val;
      int dir=0;
      snd_pcm_uframes_t frames;
      char *buffer;
      unsigned int seconds;
      unsigned int micro;
      unsigned int bufread = 0;
      unsigned int savbufread;
      int samp;

      int ret;

      snd_pcm_uframes_t avail;
      snd_htimestamp_t tstamp;


      //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);
      snd_pcm_sw_params_alloca(&swparams);

      /* 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_MMAP_INTERLEAVED);

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

      /* One channel */
      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);
    }

    //Software Parameters
    rc = snd_pcm_sw_params_current(handle, swparams);
    if(rc < 0) {
      printf("unable to determine software params %s\n", snd_strerror(rc));
    }

    /* Enable Timestamps. */
    rc = snd_pcm_sw_params_set_tstamp_mode(handle, swparams,
SND_PCM_TSTAMP_MMAP);
    if(rc < 0) {
      printf("Can't set timestamps\n");
      exit(1);
    }

    rc = snd_pcm_sw_params(handle, swparams);
    if (rc < 0) {
      fprintf(stderr, "unable to set sw 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 loop**********************************/

    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) {

        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;
        }

        rc = snd_pcm_mmap_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");
        }
        //Read timestamp
        rc = snd_pcm_htimestamp(handle, &avail, &tstamp);
        if(rc < 0) {
        printf("Timestamp Error: %d\n", rc);
        return 1;
        }

        seconds = tstamp.tv_sec;
        micro = tstamp.tv_nsec / 1000;
        fprintf(file,"%d,%ld,%lu,",seconds,tstamp.tv_nsec,(unsigned
long)avail);

        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);
}
```