Justin Gorgen
Ali Ghorbani
UCSD CSE237B Spring 2016

A Model for Active Audio Noise Cancellation

Goal

The goal of the project will be to create a model of an audio active noise cancellation system. This model will consist of two BeagleBone nodes, a master and a slave. The master node will play an audio signal from a speaker, and the slave node will attempt to attenuate the signal by transmitting a time-delayed and inverted version of the same audio signal to cancel out the incoming wave.

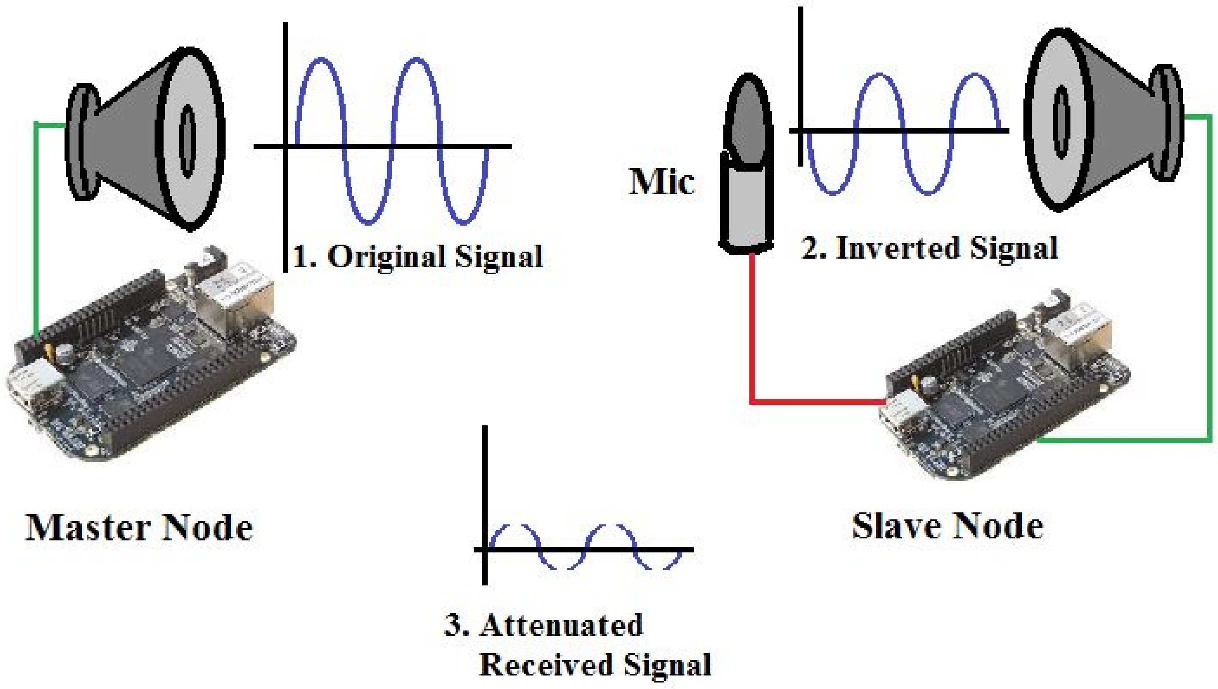


Figure 1. The original plan was to synchronize the nodes sufficiently to cancel out an audio signal over the air The slave node will try to cancel out an audio signal (1) sent from a master node by transmitting an inverted signal (2). Performance will be determined by comparing the amplitude of the attenuated signal (3) with the original amplitude.

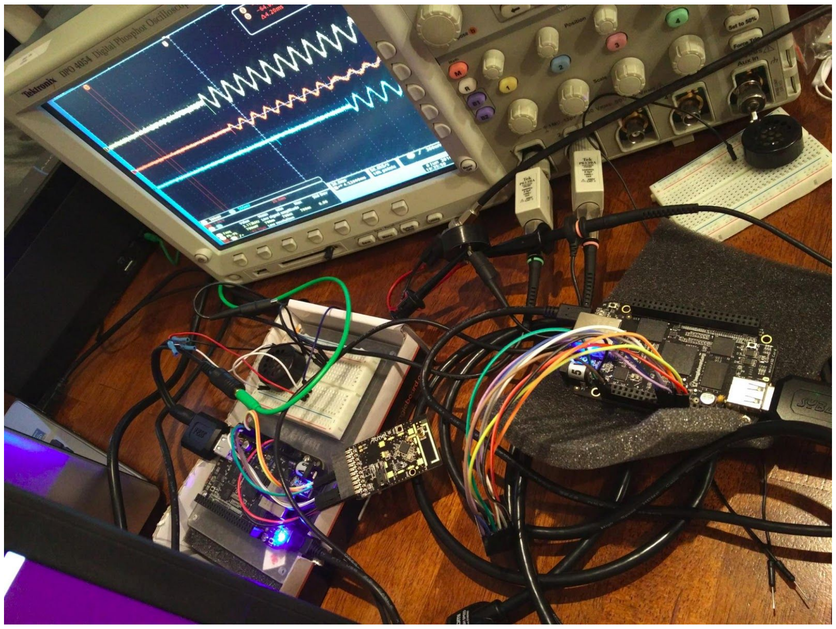


Figure 2. Setup

Software Stack

The application was implemented with Go Programming language on Ubuntu; and it constituted a Client piece and a server piece that talk to each other using tcp.

We needed a precise sleep operation hence we implemented a “precisesleep” library using C and created a C-binding for our go application so it could operate The low-level library enabled us to control sleep operations to nanoseconds.

To communicate with the audio device we used PortAudio library which has a binding for go lang. The library was used primarily to play audio files on the devices.

As part of the implementation we attempted to record tones from another party and perform real time fourier transform using a DSP library. The recording of audio and the fourier transform on an existing wav file were successful however we were not able to implement a live sensing/processing routine.

We also experimented with UDP connections, audio recording, FFT, and syscalls.

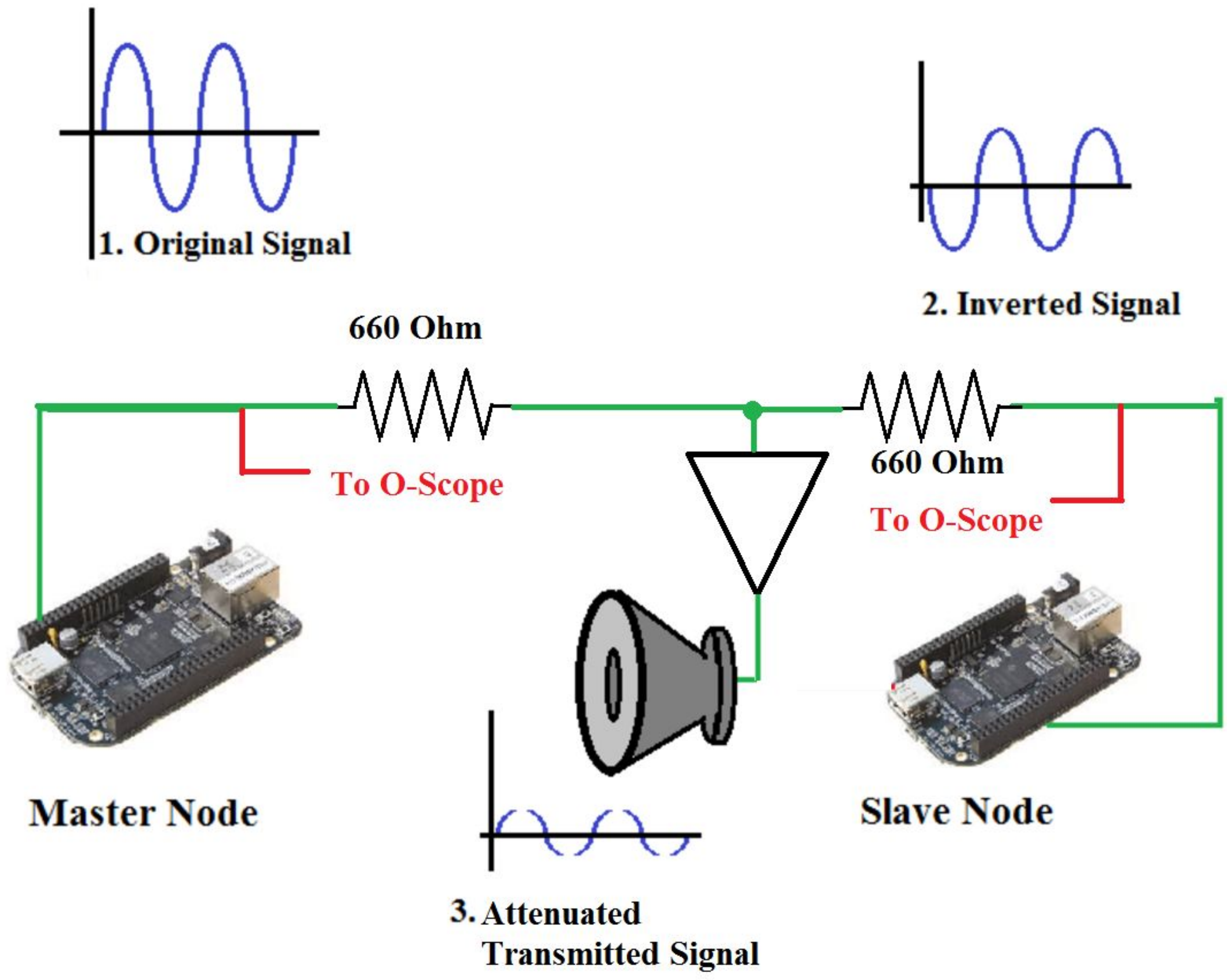


Figure 2.. The revised plan was to synchronize the nodes sufficiently to cancel out an audio chirp by electrically adding summing a signal from the slave node and master node. The slave node will try to cancel out an audio chirp(1) sent from a master node by transmitting an inverted signal (2). Performance will be determined by using an oscilloscope to compare the start of (1) with the start of (2);

Implementation Plan

During the main operation cycle of the noise cancellation scheme, the slave will alternate between measuring the incoming master signal, and cancelling the incoming master signal. This measurement state will allow the slave to estimate the time delay $\mathbf{t}_{\mathbf{ms}}$ and amplitude attenuation between the master and slave, $\mathbf{A}_{\mathbf{ms}}$. The slave will then switch to the cancellation state, where it will cancel the master signal by transmitting its own inverted signal to cancel the expected amplitude and time of arrival.

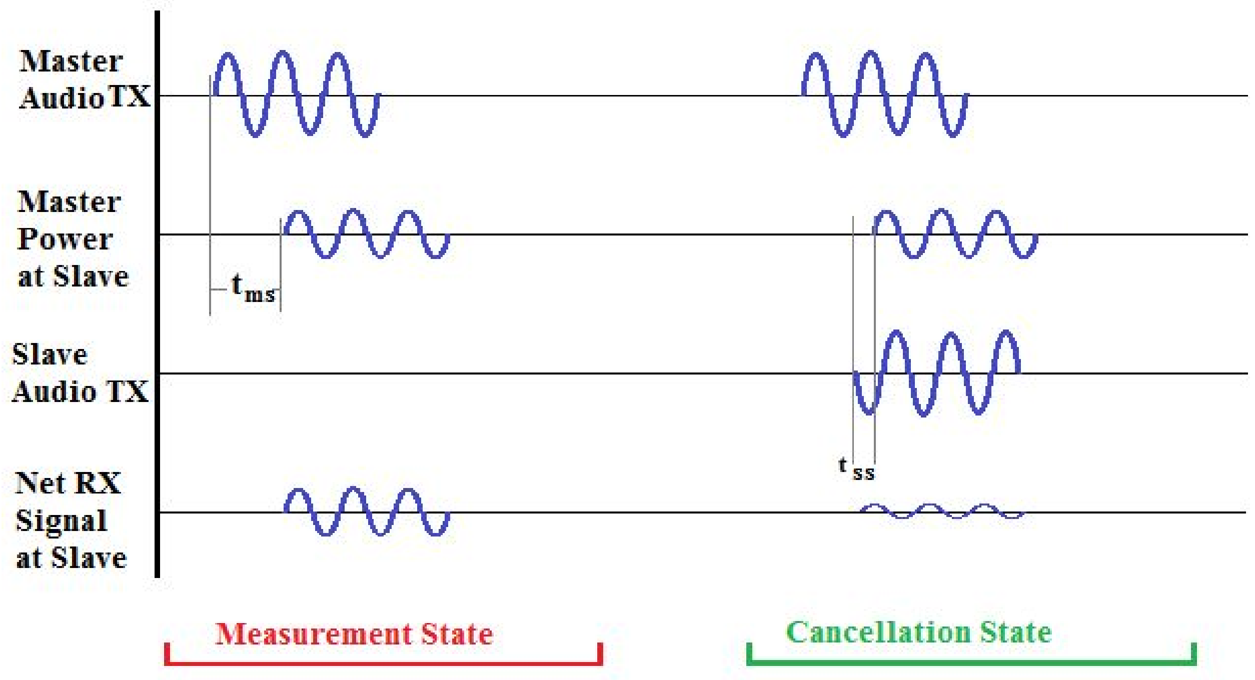


Figure 2. The master node transmits an audio signal at 1.000 second intervals. The slave node alternates between a Measurement States and Cancellation State. During the Measurement State the incoming audio amplitude $\mathbf{A_{ms}}$ and time of arrival $\mathbf{t_{ms}}$ of the received signal. The slave then switches to the cancellation state and transmits a signal that is early by $\mathbf{t_{ss}}$ with amplitude $\mathbf{-A_{ms}/A_{ss}}$

The state transition diagram for the slave node is simple, as shown below in Figure 3. At startup, the slave enters calibration. During the Calibration State, the slave transmits an audio signal of amplitude 1, and measures the resulting time of arriva, $\mathbf{t_{ss}}$, and received amplitude, $\mathbf{A_{ss}}$. This time delay and amplitude factor will be used to estimate the correct delay for transmitting the cancellation waveform during the Cancellation State.

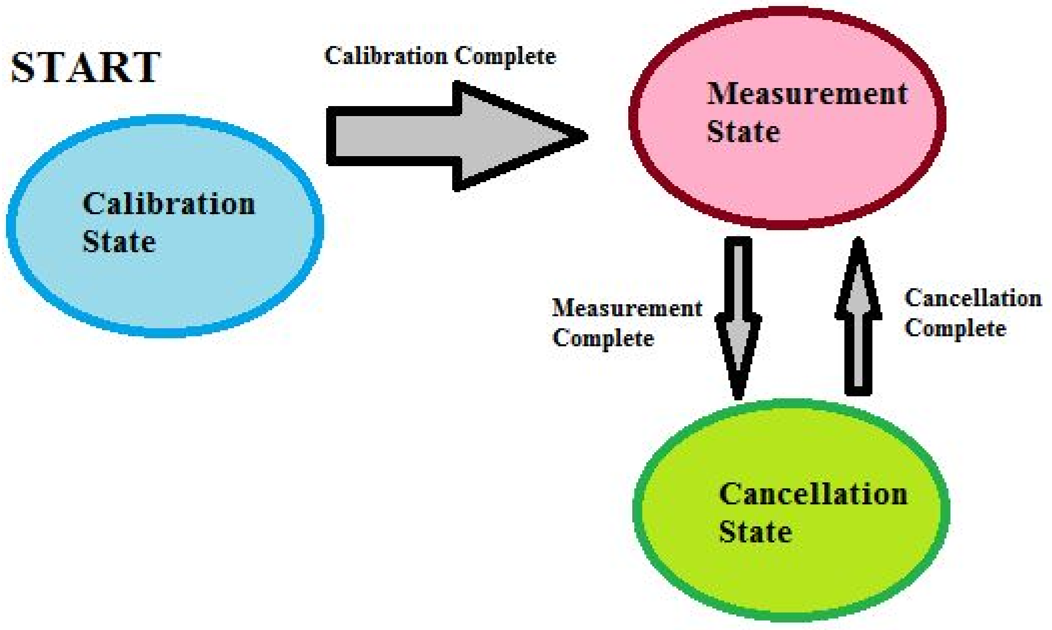


Figure 3. State Transition Diagram for the slave node.

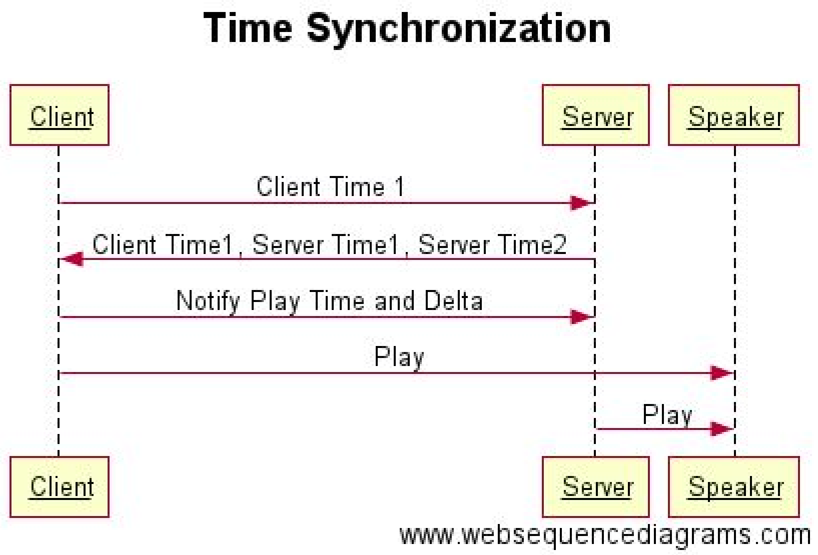
In a goroutine separate from the state transition thread, the slave will also be communicating over 6LowPan with the master to maintain synchronization. The format of the time synchronization message is shown below in Table 1.

Field Name	**Type**	**Description**
timeSent	int64	The time this current message was transmitted, in nanoseconds since the Unix Epoch
timePassed	int64	The time in nanoseconds since the last message was transmitted
lastReceived	int64	The timestamp on the last message from the communication partner
timeOfPlay	int64	The time the master will play a sound, in unix time

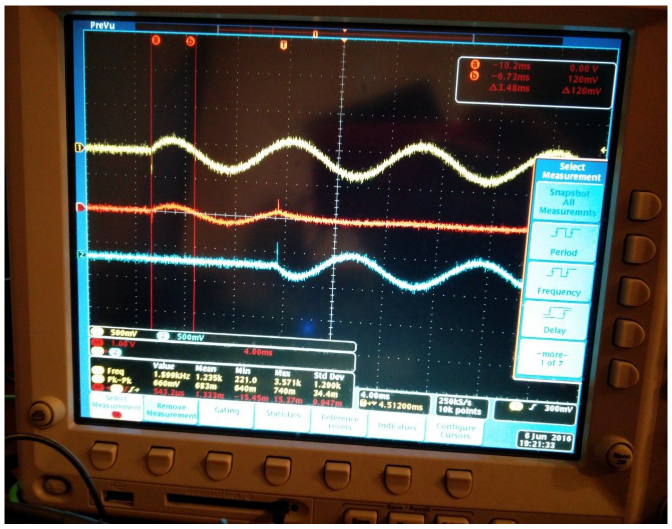
Table 1. The format of the timestamp messages used to communicate between slave and master

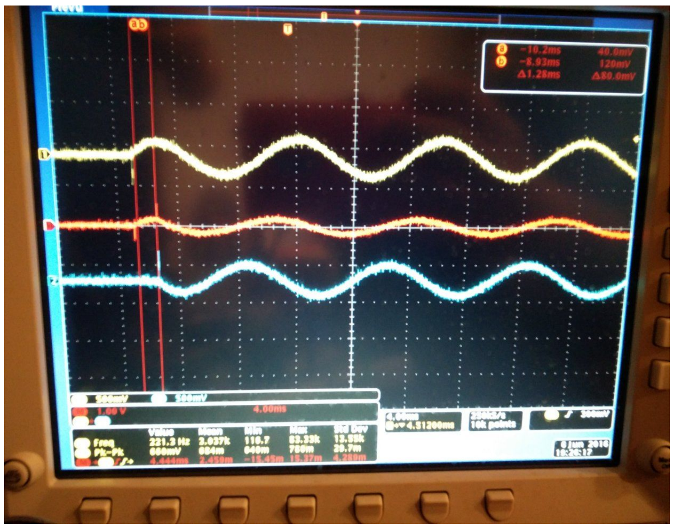
Client/Server Time Synchronization

One of the boards act as the server and the other one as the client. The client sends its time to the server, the server records the time of reception, then it generates a new time and sends all three times together to the client. The client uses the recorded times to measure the delta between the two devices, this operation is repeated 10 times and the average of all attempts is used as the reference point. To demonstrate the effectiveness of synchronization both devices attempt to play the specified frequency each time that they communicate, ideally the effect of synchronization should become more evident by observing that the cancellation becomes more effective as the synchronization proceeds. In practice, however, there were many cases that due to noise in the environment, the cancellation was not successful.


Time Synchronization
Client
Server
Speaker
Client Time 1
Client Time1, Server Time1, Server Time2
Notify Play Time and Delta
Play
Play
Client
Server
Speaker
www.websequencediagrams.com

Our attempt to observe cancellation using lower frequencies was much more consistent however due to limitations of the speaker we could not hear the low frequencies effectively hence we used oscilloscope to debug the waves. It was observable on the oscilloscope that the cancellation was consistently happening on lower frequencies, it was particularly observable that as the two devices synchronize their time and converge on an average delta, the cancellation is more effective.





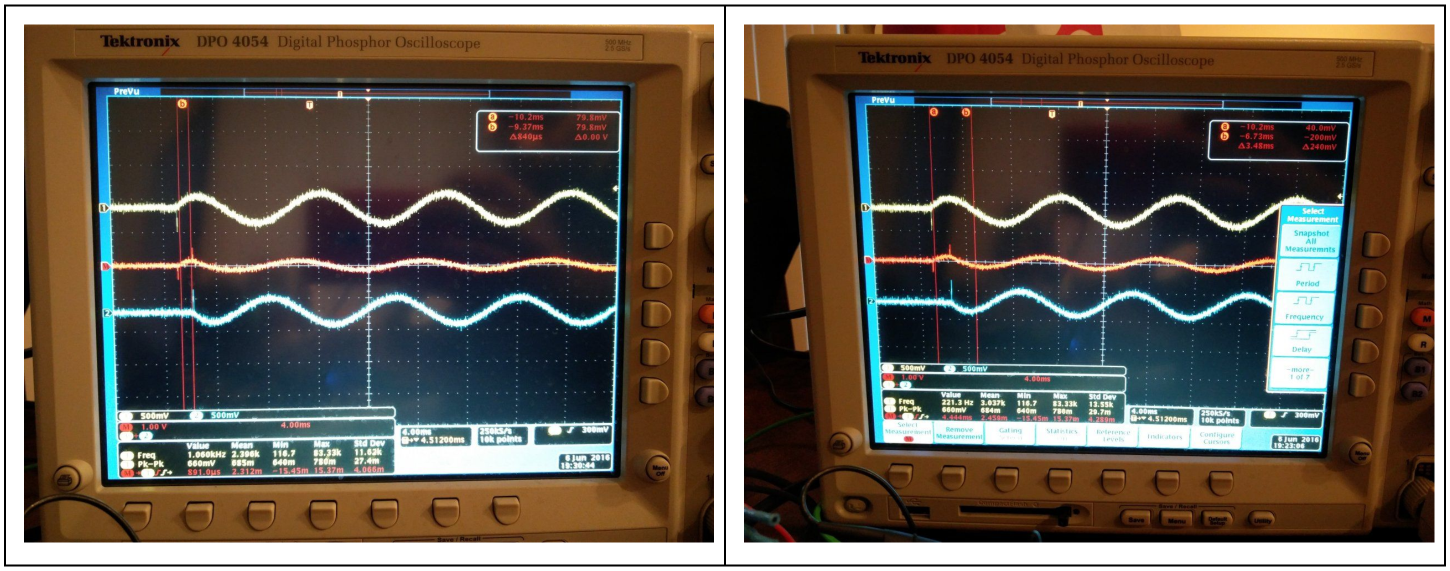

Tektronix DPO 4054 Digital Phosphor Oscilloscope
Tektronix DPO 4054 Digital Phosphor Oscilloscope

Figure 3. Successful cancellation for lower frequencies.

Timing Improvements

The earlier experiments in lab 2 determined that the total jitter was 1.04 milliseconds. Of this 1.04 milliseconds, approximately 95% of the jitter (1.0 milliseconds) was due to jitter in the TCP/IP and lower layers. Since this is outside the control of the golang environment, the only optimization here will be to switch to UDP, since this ACK-less protocol has less overhead when packet loss is not an issue.

The second optimization will be to switch from using golang's built in timers to reading the BeagleBoneBlack's hardware clock, since lab2 showed that approximately 11 microseconds was due to using golang's timer and another 11 microseconds of jitter was due to reading golang's time.Now() function.

Another improvement was to use precise sleep operation using a C implementation, the nanosecond control over the operation using clock_nanosleep function of linux was proven to be significantly more accurate than go's tickers and timers.

We tried to make SysCalls to read system timing but the jitter on the system calls was significantly higher than golang's built-in "now" calls.

Future Steps

Replacing PortAudio with a lower level library: PortAudio was proven to be inefficient with a large overhead, ideally we should have communicated directly with the audio device for playback.

Go language overheads: We did not do any optimization on the go language runtime, for example we could have turned off garbage collection and we could have benchmarked runtime delays by comparing the same operation with a C based application, both as client and as server.

Justin Gorgen
Ali Ghorbani
UCSD CSE237B Spring 2016

Timing Results

We detected overall jitter of 3.5 ms , we assumed that ⅓ of the inverse of this value, which is 100 Hz, should consistently yield cancellation however, we were not able to reproduce it during the demo, we realized that the actual value must be 1/9 times of the inverse of the jitter which is around 30Hz, something that we could have potentially demoed using an oscilloscope.