

# **CS176b Deliverable C: Audio Messenger**

William Chen, Chien Kai Wang, and Brian Wan

## **I. Introduction**

For our project, we created a messenger application with Python. This application is capable of two forms of communications: text messenger and voice over IP. The user is able to switch back and forth between text and voice as they choose for a better chatting experience, as compared to only text or only voice. On top of chatting, we also implemented security, in the form of encryption and decryption. The project explores the topic of streaming, voice over IP, and internet security.

We chose to do a text messenger because it was the best and most well-known example of computer network communication. We added voice over IP to see how the concepts would hold with voice packets and learn how the network would affect voice quality. We took on security along with it, because in the modern age, communication packets could be easily intercepted. Adding security to a chat client should be a standard.

A majority of the programming was performed on Mac OS X and Ubuntu 14.04. However, it seems like there are inconsistencies between PyAudio for Mac and PyAudio for Ubuntu, and so, voice over IP between the two operating systems does not work. Text still does, however.

## **II. Project Design**

The result of our project resulted in a Skype-esque application that ran from the command line. The application only needs two files to run: a server and a client. Client only needs the IP address of the computer running the server file. For simplicity's sake, the port is set within the program, but it can easily be changed with a simple text editor. This single server would handle both the text packets and the audio packets.

At this time, the server can handle up to five different connections to clients, but this could also be easily changed. The maximum number of client connections the server can handle

depends on the operating system and hardware running the server, but it is usually five connections, and therefore, we set it to listen to five connections to prevent any problems. Any messages sent from a client would first go to the server, and the server would broadcast those messages to all the other clients.

When a user first connects, the client is set to text mode by default. If multiple clients connects to the server, the 'select' function plays a key role, because without it, it would be impossible for the clients to receive messages from the standard input and the server simultaneously. It selects the stream that is ready instead of waiting for a specific input from the user.

However, due to this select function, our application only works with Linux and Mac operating systems. The 'select' function used to broadcast does not work with Windows's version of the socket module for Python (1), and the work-around options were much too complicated. We would had to have create two separate servers, and those servers would need to communicate with each other, and troubleshooting that along with voice and security would have been extremely difficult.

When a user at one of the client decides to use voice to communicate, text messages are temporarily disabled while the user talks. The client and server programs will automatically switch back to text after silence is detected. Due to variance of microphone quality between our computers, the threshold used to determine silence is set to a low bound for simplicity in debugging. Silence could be simulated by muting the microphone in the computer's system settings.

Most of the Python modules we needed to use were already provided with the 2.7.9 version of Python. The only module we had to go install separately was 'PyAudio'. We decided to use Python version 2.7.9 to ensure compatibility with PyAudio. PyAudio is the module that handles receiving and playing the audio streams received from the user's microphone.

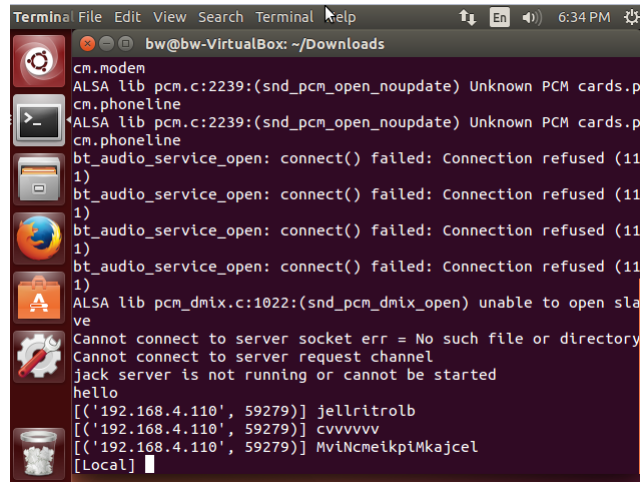
In class, it was mentioned that most voice over IP applications are using TCP connections, because the quality of the internet connection as a whole is stable to the point where TCP failures are rarely encountered. For this reason, we decided to use TCP sockets to try and emulate what would be found outside the scope of our class. However, we implemented our own send and

receive functions, different from what was provided by the socket module, in case the messages sent were too big and needed to be broken up into separate packets.

To switch to voice communication, the user types '[audio]' into the prompt and strike the 'Enter' key twice. The application is switched into voice mode. This is made possible by sending and receiving 'dummy' variables and values that the client and servers use to trigger our 'if/else' cases that switch the application into voice mode. When silence is detected by the client that started the voice communication, the dummy values change, the application switches back into text mode.

We have a function "is\_silent" that takes the voice that will be sent, and scans it for the maximum "intensity" of the sound. If the maximum intensity is less than the set threshold, currently set to 600, then it is perceived as silence (2).

The final part of the project is the security. We implemented a password-based substitution cipher with predetermined special characters. While substitution ciphers are relatively easy to break, this one is significantly harder to break because of the password system. All encryption is client-based, so the server doesn't know what message is sent. When starting the client, the user is asked to rearrange some words to form a password. This is the key to the substitution cipher. This way, if one client submits a different password than another, they will not be able to understand each other; all clients need to know the same password. Both lowercase letters and uppercase letters are encrypted with both being encrypted differently from the other. The encryption uses Python dictionaries for simplicity and effectiveness with the initial setup using lists of tuples. Because audio packets work slightly differently than text, we had problems trying to encrypt the voices, and using a single bit, we determined if the message being sent is either voice or text. The encryption scheme we use only encrypts strings. Initially, an attacker will only know the length of the message and whether or not it is audio. The symbols like periods and commas will always be the same, so he can figure that out relatively simply. However, since the password, and hence, the cipher, can change every time, the attacker will have a harder time decrypting the messages across sessions.



```
Terminal File Edit View Search Terminal Help
bw@bw-VirtualBox: ~/Downloads

cm.modem
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
cm.phoneline
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
cm.phoneline
bt_audio_service_open: connect() failed: Connection refused (11
1)
bt_audio_service_open: connect() failed: Connection refused (11
1)
bt_audio_service_open: connect() failed: Connection refused (11
1)
bt_audio_service_open: connect() failed: Connection refused (11
1)
ALSA lib pcm_dmix.c:1022:(snd_pcm_dmix_open) unable to open sla
ve
Cannot connect to server socket err = No such file or directory
Cannot connect to server request channel
jack server is not running or cannot be started
hello
[['192.168.4.110', 59279]] jellritrolb
[['192.168.4.110', 59279]] cvvvvvv
[['192.168.4.110', 59279]] MviNcmeikpiMkajcel
[Local]
```

Figure 1: If the wrong password is given

### III. How to Use Audio Messenger

In order to use Audio Messenger, PyAudio must first be installed. The PyAudio module can easily be found as a simple download on the main PyAudio website, <http://people.csail.mit.edu/hubert/pyaudio/>.

As mentioned before, our application uses Python 2.7.9, so we need to download PyAudio for Python 2. On the Mac, ignore the download where it says “For Python 3 support” and click on the download link. After downloading, double click the file and a few more instructions come up, and when we follow it to the end, PyAudio would be installed.

**Apple Mac OS X**  
Includes: PortAudio v19 [SVN r949]  
Requirements: Intel Mac running Apple Mac OS X 10.7+. For Python 3 support, first install [MacPython 3.3](#).  
Download: [PyAudio for Mac OS X](#)  
md5sum: 330133154e83f4d778ff04ac327f47f2

**Debian/Ubuntu**  
To install, download the package and then run:  
% dpkg -i python(,3)-pyaudio\_0.2.8-1\_(i386,amd64).deb  
Download: [PyAudio for Python 2 \(i386\)](#)  
md5sum: 1e6bccc1db76c6bd386fa77eafef3ae  
[PyAudio for Python 3 \(i386\)](#)  
md5sum: 13d76cc7a9f58a1f022c104770cf8f5d3  
[PyAudio for Python 2 \(amd64\)](#)  
md5sum: f8f3b06c8c81023e024403054c48f9e  
[PyAudio for Python 3 \(amd64\)](#)  
md5sum: 3803fe0b10c1907cda8fb0853972810a6

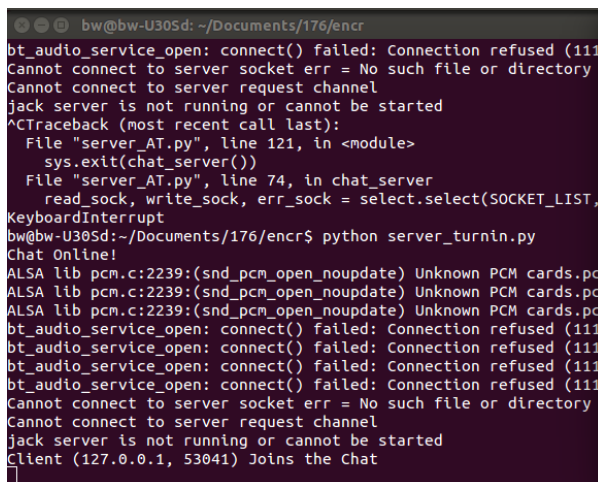
Figure 2: Downloading PyAudio

On Ubuntu and Ubuntu variants, there is a separate download, titled “Debian/Ubuntu”. It also depends on whether or not the operating system is 32-bit or 64-bit. 32-bit operating systems

would download “PyAudio for Python 2 (i386)” while 64-bit operating systems would download “PyAudio for Python 2 (amd64)”. The PyAudio website also gives the command to install PyAudio. Alternatively, double clicking the file on Ubuntu will open the Ubuntu Software Center to the PyAudio page, and clicking the install button on the right side would install PyAudio onto the computer.

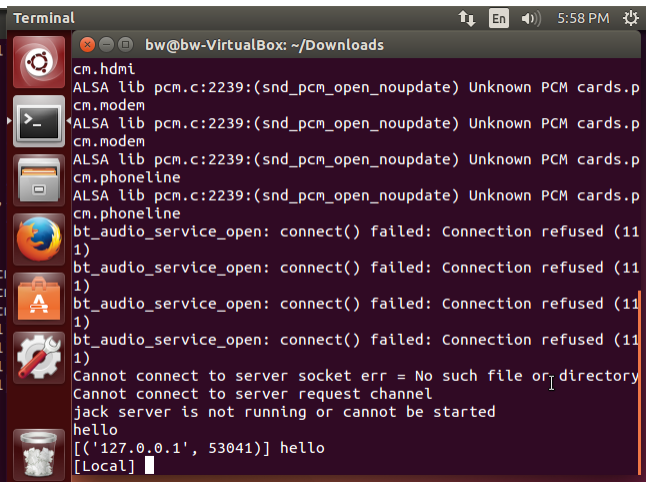
PyAudio is now installed and we are ready to use Audio Messenger. Audio Messenger runs from within the command line, so we must open a terminal, and cd to the folder containing the Python files. The server must be started first. Before we do that, we must use ‘ifconfig’ to get the current IP address of the computer. The clients must use this IP address to connect to the server. To start the server, use the command ‘python server\_turnin.py’. It automatically uses ‘localhost’ as its IP address, and the port is automatically set to 9001, though as said before, this can easily be changed. On Ubuntu, a large list of error messages appears, but these can be ignored. The server is still working fine. These messages do not appear on Mac.

The clients can be started with the command ‘python client\_turnin.py X’, where ‘X’ is the IP address found from the ‘ifconfig’ command.

A terminal window titled 'bw@bw-U305d: ~/Documents/176/encr' showing the execution of 'python server\_turnin.py'. The output displays several error messages related to ALSA and bt\_audio\_service\_open, followed by a successful connection from a client at IP 127.0.0.1.

```
bw@bw-U305d: ~/Documents/176/encr$ python server_turnin.py
bt_audio_service_open: connect() failed: Connection refused (111)
Cannot connect to server socket err = No such file or directory
Cannot connect to server request channel
jack server is not running or cannot be started
^CTraceback (most recent call last):
  File "server_AT.py", line 121, in <module>
    sys.exit(chat_server())
  File "server_AT.py", line 74, in chat_server
    read_sock, write_sock, err_sock = select.select(SOCKET_LIST,
KeyboardInterrupt
bw@bw-U305d:~/Documents/176/encr$ python server_turnin.py
Chat Online!
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
bt_audio_service_open: connect() failed: Connection refused (111)
bt_audio_service_open: connect() failed: Connection refused (111)
bt_audio_service_open: connect() failed: Connection refused (111)
bt_audio_service_open: connect() failed: Connection refused (111)
Cannot connect to server socket err = No such file or directory
Cannot connect to server request channel
jack server is not running or cannot be started
Client (127.0.0.1, 53041) Joins the Chat
```

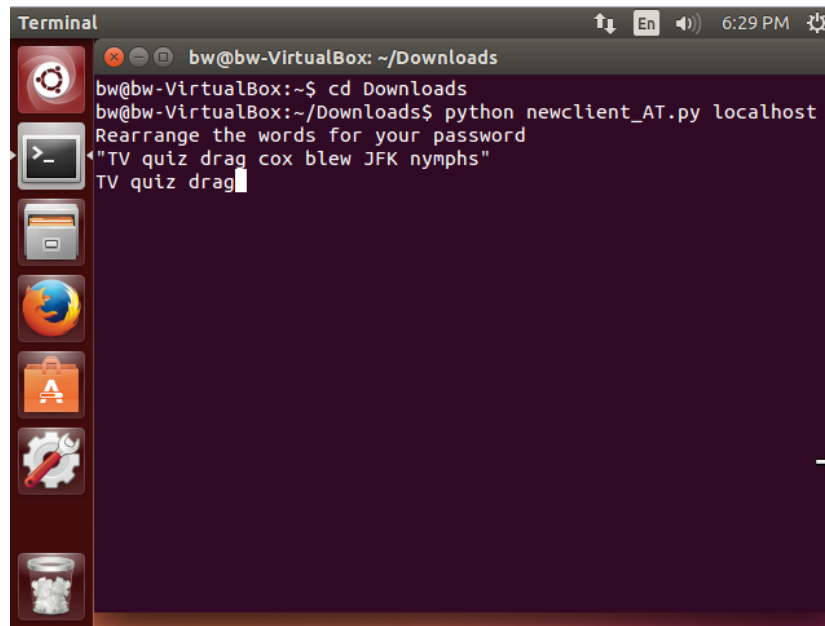
Figure 3: Server Error Messages

A terminal window titled 'bw@bw-VirtualBox: ~/Downloads' showing the execution of 'python client\_turnin.py'. The output displays several error messages related to ALSA and bt\_audio\_service\_open, followed by a successful connection to the server at IP 127.0.0.1.

```
bw@bw-VirtualBox: ~/Downloads$ python client_turnin.py
cm.hdmi
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
cm.modem
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
cm.modem
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
cm.phoneline
ALSA lib pcm.c:2239:(snd_pcm_open_noupdate) Unknown PCM cards.p
cm.phoneline
bt_audio_service_open: connect() failed: Connection refused (111)
bt_audio_service_open: connect() failed: Connection refused (111)
bt_audio_service_open: connect() failed: Connection refused (111)
bt_audio_service_open: connect() failed: Connection refused (111)
Cannot connect to server socket err = No such file or directory
Cannot connect to server request channel
jack server is not running or cannot be started
hello
[('127.0.0.1', 53041)] hello
[Local]
```

Figure 4: Client Error Messages

It will then ask you to rearrange the words in a sentence. This is used to create the substitution key. The password is case and space insensitive. Remember that in order for multiple clients to successfully decipher each other’s messages, all clients must have the exact same password. Now, the client will attempt to connect to the server that is running.

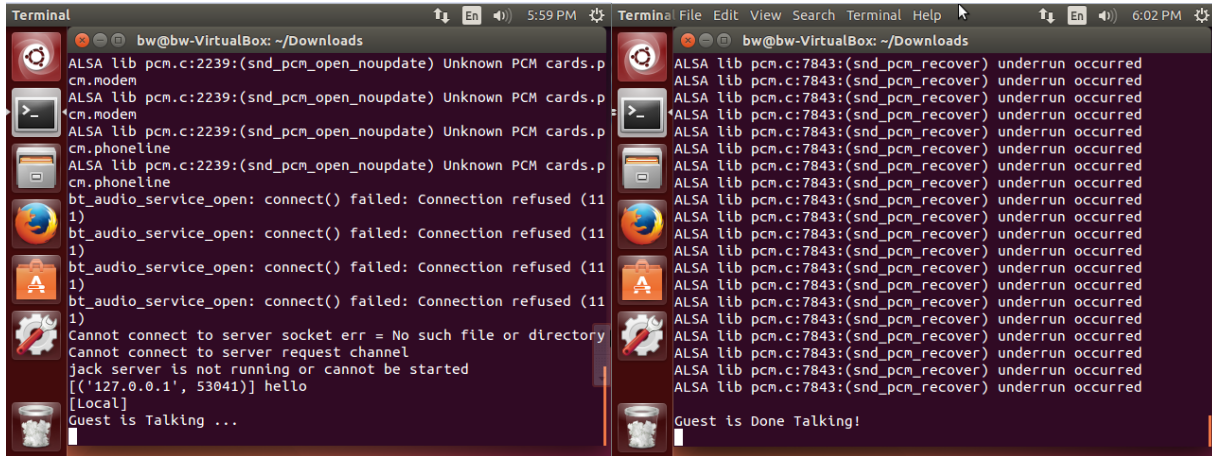


```
Terminal
bw@bw-VirtualBox: ~/Downloads
bw@bw-VirtualBox:~$ cd Downloads
bw@bw-VirtualBox:~/Downloads$ python newclient_AT.py localhost
Rearrange the words for your password
"TV quiz drag cox blew JFK nymphs"
TV quiz drag
```

Figure 5: Asking for a password

If the connection is successful, on the server terminal, you will see that a client has connected, and its IP address is given. In the client terminal, there will be error messages again, but once again, these can be ignored. If multiple clients join, then the IP addresses will also show up on the other clients when another joins. Text chat is as simple as typing and striking enter. The chat log on the terminal will display the IP address of where the message came from. Blank messages can also be sent with no problems.

To send voice, on one of the clients, type '[audio]' and strike the 'Enter' key twice, and the application switches to voice mode. The clients that are not using voice will display the message 'Guest is talking...' and those clients will have the chat disabled. When silence is detected, audio mode stops and all other clients will show the message 'Guest is Done Talking!' and the application switches back.



Figures 6 and 7: When a Client is using Voice

## IV. Implementation

The core backbone of the server and client were modeled after K Hong's multi-client text chat (3). To begin, the server immediately sets a variable 'msgType'. This variable will be used to help the server switch between voice and text modes. Then we define our custom send and receive functions, MsgSend and MsgRec, respectively. MsgSend uses two parameters: the socket to send to and the message itself. MsgSend will send the msgType variable, which will be used for all the other clients to switch into voice mode. It then determines the length of the message, and sends the msgType, length, and the message itself. If the message doesn't send, then the socket is removed from the list of all sockets.

MsgRec only uses a socket as its parameters. It receives the msgType that was sent with MsgSend, and determine the length of the message. If the message is less than 1024 bytes, it will continue receiving and append everything received into the variable 'msg' and returns it.

After defining these functions, we import all the modules needed for the application to run. We import the pyaudio, sys, socket, and select modules. As said before, PyAudio is the only module not already included with Python.

Next, we define the variables 'host' and 'port', used for the connections. Host is set to localhost and port is set to 9001 by default. 'SOCKET\_LIST' is a list that will hold all the client connections. Then, we create the actual server.

We set the server to use TCP sockets. We also set the server to reuse the port if the application were to be stopped. We bind the port, and if it is not able to be bound, the server would stop. We add the server's socket to `SOCKET_LIST`. We create an instance of PyAudio, which will be used to stream the voices. The 'stream' variable used to stream is set up like the examples on the PyAudio web page. Then, we set the server to listen for up to five connections, and create a variable that will check for voice. The server then chooses an input stream that is ready to use.

For each socket that is trying to connect, the server will accept and print in the terminal that a client has connected, along with its IP address and socket number. It would then broadcast this same information to all other clients currently connected. Then, the server begins to receive messages from the clients. If a command to use voice was not given, the `checkAudio` variable remains at 0, or else it will switch to 1. If it's just plain text, the server sends the text message along with the sending client's IP address to identify where it came from. If the command to use voice was given, the voice data is simply relayed without formatting. If the server detects a broken socket, that socket will be removed from `SOCKET_LIST`. This will continue in a 'while 1' loop.

We then define our 'sendtochat' function, which is the function used to broadcast messages to all other clients. The client that sent the original message is, of course ignored. It uses our custom 'MsgSend' function written earlier. Lastly, we use 'main' to start the server.

On the client, we begin by importing all the modules we need to run the application. Once again, PyAudio is the only module not included in Python by default. We define a few variables used for checking for the types of messages and to check if messages were received. Then, we create lists of characters that we will use for our encryptions. We define the same 'MsgRec' and 'MsgSend' functions as the server, except we have encryption and decryption added, and then we define our encryption and decryption functions, 'encrypt\_sub' and 'decrypt\_sub' respectively, both only requiring the message to be encrypted. As mentioned before, only text strings will be encrypted and decrypted.

Then we create the chat client. To begin, it checks if an IP address was given. This IP address is used to connect to the server, and without it, the client automatically closes. Then it



asks you to create a password by rearranging the words given, and based on the ordering of the words, the ciphers are created. It will also check if the length of the password is correct, and if it isn't, the client will close. After the ciphers have been made, the client will attempt to connect to the server.

We create an instance of PyAudio, an array to hold the audio, and a counter that is used to detect silence. If the client receives a text message, the client will print it to the terminal. To prevent errors, it will clear the terminal input line as well. If no messages are received, then the client can type and send text messages. If it is text messages, then the messages are sent with our custom send function, which will also encrypt the message. If the command to use voice is given, we reset our check, type, and counter variables to prepare for audio.

If the client is receiving audio, a stream is opened to output the voice sent. For audio streaming on the client side, depending on the variable receive, the stream will be defined as either input or output with frame rate equals to 44100, frames per buffer equals to 8192, and stream format of pyaudio.paInt16 ( We tested with many different frame rates and frames per buffer combination and some frame rates with lower frames per buffer number gave us input stream errors based on different operating platforms). The Client will continue outputting audio until an 'endofaudio' string is detected. If the client is sending audio, the stream is opened to input voice. The stream that is run through our silence detection function 'is\_silent' to detect for silence before sending. After 40 loops of silence, the client automatically sends an 'endofaudio' string and all our checking variables are reset. This is all wrapped within a 'while 1' loop.

At the end, we define the 'is\_silent' function, which takes voice data as its parameter, and the 'main' function, that runs the client.

## **V. Analysis**

In our tests, we never had dropped packets. We tried on separate home networks, on the same home networks, and using the UCSB Secure network. The text messages encrypted and decrypted with no problems, and the only issue with the audio is the slight delay.

We found that for an application such as ours, a half-duplex communication system, using a centralized system was easier to implement. We tried to do a decentralized system, where the clients connect to each other. We used multiprocessing so that the audio would work in the background while the handling of text messages operated in the foreground. However, we were unable to get the clients to connect to each other. The clients would crash before the connections could be made. We think there is a small detail about multiprocessing and multithreading in Python that we are missing. We have provided the python files demonstrating what we tried to do with this idea. The codes are in the 'Trial Code' folder, named 'client\_audio\_multi' and 'client\_audio\_multi2'.

Our first plan for security was to use RSA public key and private key encryption. We wanted to use this form of encryption, because it is what is most similar to what is being used in real world applications. However, with multiple clients, the code got messier and messier, to the point where debugging was much too difficult. We were able to get to the point where one of two things was happening: the messages were being encrypted, but not successfully decrypted, or the messages were not encrypted to begin with, but was decrypted anyway, into messages that made no sense. We have also provided the code to demonstrate our work with RSA encryption. It is also in the 'Trial Code' folder, named 'client\_RSA'.

When text messages were being sent, there was not much delay between striking the 'enter' key and for the text to show up on the other clients. It was very quick, and the slight delay was reasonable. It is due to the fact that the characters in the string must be substituted and rewritten, due to our encryption scheme. The same can be said about decryption. In the case of voice, however, there was a noticeable delay. The delay doesn't destroy the user experience, per say, but it is noticeable. This is due to having to fill the packet with voice data.

When voice is being sent, there is a clicking sound in the background. The overall audio is still understandable. This started to happen after we implemented the security. This is because we have to do some extra processing in our 'MsgRec' and 'MsgSend' functions. They need to first determine if it's audio or text, and during that time, for a brief moment, nothing is being sent after the previous voice packet was finished playing. As we learned in homework 2, when there is a sudden drop or a sudden jump in sound amplitude, then there will be clicking sounds.

Voice crossing over operating systems was a problem with our application. When it was tested between two separate Ubuntu systems, voice was sent back and forth just fine. When it was connected to itself, via localhost as its IP and the use of multiple terminals, voice worked just fine. The same can be said on the Mac system, when we connected to itself. We believe this is due to the inconsistencies between PyAudio versions for Ubuntu and for Mac. Without knowledge on how exactly PyAudio works and how Ubuntu's and Mac's operating systems work, it's hard to troubleshoot this issue without breaking something else.

## **VI. Future Works**

For what we wanted to deliver at the start of the project, we finished most of what was on our checklist. The only thing we were not able to do was create a graphics user interface. Our security was done, but it wasn't very well done. We implemented something simple that would never be used in the real world due to it being a substitution cipher. Although it is a password protected substitution cipher, trespassers would be able to break the cipher in finite time.

In the future, we could spend more time trying to debug the RSA encryption and get it to work. It is the most robust security system and most widely used, so RSA encryption should be used for this project. We would also create a graphics user interface to improve user experience. The GUI would make using the application much more intuitive. Another fix would be the clicking sound during the playing of the audio. This would get annoying really fast. It could be solved with a buffering system.

Also, fixing voice over different platforms would be done in the future. It would be nice to make our application work regardless of the operating system. This could be solved by looking into a different audio library for Python. Finding a workaround for the 'select' function on Windows would also be nice too.

**Sources:**

- (1) <https://docs.python.org/2/library/select.html>
- (2) <http://stackoverflow.com/questions/19070290/pyaudio-listen-until-voice-is-detected-and-then-record-to-a-wav-file>
- (3) [http://www.bogotobogo.com/python/python\\_network\\_programming\\_tcp\\_server\\_client\\_chat\\_server\\_chat\\_client\\_select.php](http://www.bogotobogo.com/python/python_network_programming_tcp_server_client_chat_server_chat_client_select.php)