Realtime Steganography within Audio Streaming Application

Comp 8047 – Major Project

Ryan Eakin – A01048229
5-30-2022

Table of Contents

1. Introduction

1.1. Student Background

My name is Ryan Eakin. I started my education at BCIT in 2018 as part of the CIT Diploma Program. After graduating I decided to complete my degree and continued into the BTECH program in 2020. As part of the diploma program, I previously participated in the ISSP or Industry Student Sponsor Project Program twice.

My first project was working with Earl's restaurant creating a menu management application that was primarily using the google web application services along with the creation and development of the database and user application.
The second project I worked on was a mobile and web application that was for SAR (search and Rescue) Technology. The mobile application was created as a proof of concept of a real time photo recognition to find signs or abnormalities that could lead to a missing person while the web application involved photo segmentation the be sent to an external API for processing.

1.2. Project Description

This project is for the BCIT BTECH's project course COMP 8045/8047.

This project is an application that is utilizing an audio stream to hide information within the audio data. The application works like Spotify or apple music where a user connects, and a song starts playing. Within this application the client does not decide what music to play but the server will have a predetermined playlist of songs that it plays.

1.2.1. Essential Problems

The problem I am attempting to solve is to improve the steganography techniques and uses in a Realtime environment. With modern technology stealing and reusing music that people do not have ownership of is a growing problem and with the use of this or an application following this methodology a way to "watermark" the audio being transmitted is created. This watermark would be hard to detect as it is hidden in the audio data but using a decoder the ownership could be revealed.

1.2.2. Goals and Objectives

The goals and Objectives of this project was to create a prototype application that would host an audio streaming service that would be playing music that any listener using the client application would be able to connect with. The objective of the application was to successfully hide a message into the audio stream and determine how much data can be hidden in the audio while keeping the message undetected to the listener.

Firstly, the server provides and sends out the audio data to each connected client. The design of this server is following the methodology taught in the COMP 8005 class using the Epoll API to create a more scalable server that can handle high traffic. Then the server begins to change predetermined bits in the audio data that would have a lower effect on the audio quality.

Secondly, there is the client devices. The client can connect to the server and playback the audio data that they receive. The data is stored to a buffer to allow for the playback to be smooth and reduce the chance that the audio stutters. The hidden message will be extracted from the audio data upon receiving the packets but will still be played by the audio stream.

2. Body

2.1. Background

I originally found this topic to be quite interesting and saw a potential that I could combine it with something that I love, music. The main idea for this project came when hearing about steganography practices in the COMP 8505 class and was curious if it was possible to implement this into other types of media such as audio.

When I originally searched for projects like this and steganography in audio has been done but unreliably using MP3 format files and in a static environment where the file is manipulated and then the entire file is sent. This project differs in that the environment is a real time stream, so the transmitting of data is constant to the client devices. Along with this instead of using an mp3 format for the audio files, the application uses Vorpis encoded OGG audio files which is what applications like Spotify uses.

In this project I attempt to be able to use steganography techniques in a real-time environment which in my searching has not been attempted before. Normal steganography techniques include taking an input file and encoding the file with hidden information then being able to send out the modified file. This project differs in that instead of taking the entire file the audio input is instead split into manageable chunks that are being fed into a buffer to be sent to all connected users. These chunks will also then be spliced with another buffer containing the data that is being hidden inside each chunk of data being sent to connected users.

2.2. Project Statement

The application should be able to hide a message from the server into audio data that is then broadcasted to client devices without being detected.

2.3. Possible Alternative Solutions

The alternate solutions to use a blocking API for handling connections that would be slower and would not be able to keep up with the amount of data needing to be sent by the server. Along with this the way to make this work would create a large amount of overhead on the server as a new thread would be needed for each client.

The use of UDP instead of TCP for the transfer protocol would make the issue of speed much easier for the application. The issue would be that if packets are received out of order then that would effect the playback along with if part of the message being hidden is lost then the application failed in it’s purpose.

An alternative approach to the steganography problem would to be changing the file being sent as the amount of data to be hidden changes. The issue with this approach would be that it would be quite CPU intensive and most likely would affect the scalability of the server as more power would need to go to the management of the file then the connections. This would also make the real-time more superficial as the song would not be able to be modified once it has started.

2.4. Chosen Solution

The chosen solution is to use Python 3 to create the client and server applications. I chose this as it is the environment that I am the most comfortable with using along with having the libraries needed to be able to quickly modify and play the audio data.

The choice to use an epoll API means that the server is unable to run on windows devices and requires a Linux device to act as the server. I chose this because the stream requires to be able to send a high number of packets and handle many simultaneous connections.

The choice to use TCP over UDP for this project was because of how the server is hiding parts of a message in the packets if a part of the message was lost then the purpose of the application would also be lost. I chose TCP so the packets while slower then UDP would ensure that the entire message arrives at the client devices.

2.5. Details of Design and Development

2.5.1 Data Flow Diagram

Client

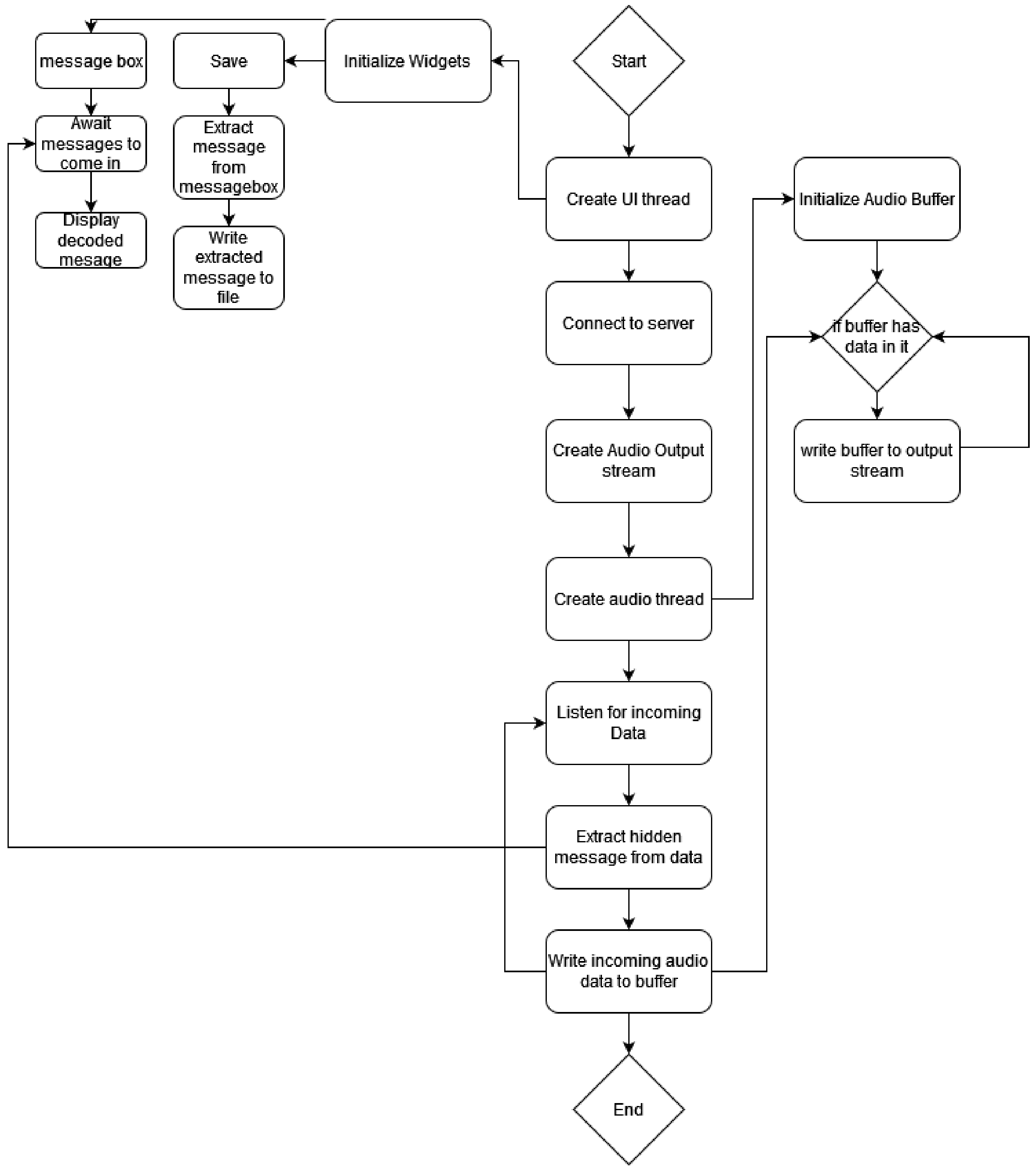
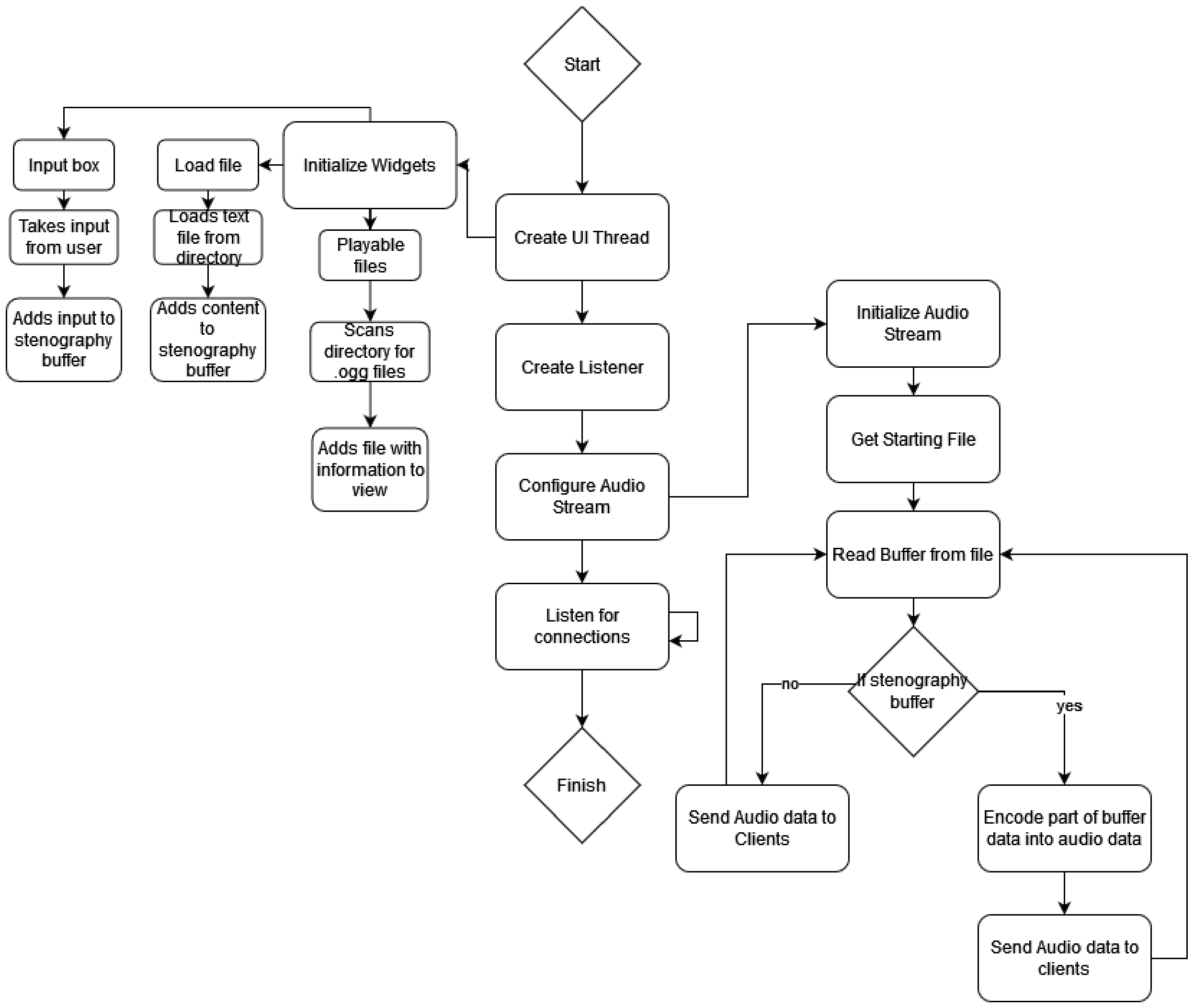

Start
Create UI thread
Initialize Widgets
message box
Save
Await messages to come in
Display decoded mesage
Extract message from messagebox
Write extracted message to file
Connect to server
Create Audio Output stream
Create audio thread
Initialize Audio Buffer
if buffer has data in it
write buffer to output stream
Listen for incoming Data
Extract hidden message from data
Write incoming audio data to buffer
End

Figure 1 Client Data Flow Diagram

The above diagram shows the architecture and data flow of the client device. This architecture allows for the program to process and play audio simultaneously. The use of a buffer allows for a smoother playback in the stream because a portion ahead of the currently played section is already processed and ready to be played. Doing this will also help if there is an error in the TCP stream and the speed that the client receives data fluctuates.

Server



Start
Input box
Load file
Initialize Widgets
Create UI Thread
Takes input from user
Loads text file from directory
Playable files
Adds input to stenography buffer
Adds content to stenography buffer
Scans directory for .ogg files
Adds file with information to view
Create Listener
Initialize Audio Stream
Get Starting File
Configure Audio Stream
Read Buffer from file
Listen for connections
no
If stenography buffer
yes
Finish
Send Audio data to Clients
Encode part of buffer data into audio data
Send Audio data to clients

Figure 2 Server Data Flow Diagram

The above diagram shows the architecture and data flow of the server. The server works by initializing connections using the epoll API and then stores the connection to broadcast the audio data to all clients. Upon startup the server will initialize the GUI for the application and will display an input box for manual entry, a button to send data, a button to load a file, and a tree view widget that scans the directory the application is running in and displays all OGG files it can find. The audio stream works by initializing the OGG file and creating a buffer for it to read from. When it reads from the buffer it will attempt to take a part of the global string which is the steganography buffer and will encode it into the audio data with a key before broadcasting it to clients.

2.5.2 Pseudo Code

```
AudioStenography():
    File=currentlyplayingfile
    buf2=characterbuffer
    while true:
        buf1 = file.getBuffer(BUFFERSIZE)
        data = np.frombuffer(buf1, dtype=np.int16)
        for i in range(0,len(buf2)):
            data[i]=[buf2[i],key)
        sendabledata = pickle.bumps(data)
        send sendabledata to all connected clients.
```

The audio Steganography function works by first finding the current file that is being played the file is then converted to a buffer that is being progressed each iteration the section of the buffer that is being sent is converted to a NumPy array and then spliced with the array containing the ascii values the amount of data being spliced can be changed as if less data is being added the time to send all the data increases but the detectability decreases. After this happens the array is pickled meaning that it is converted into an object to be sent as an array cannot be sent through packets.

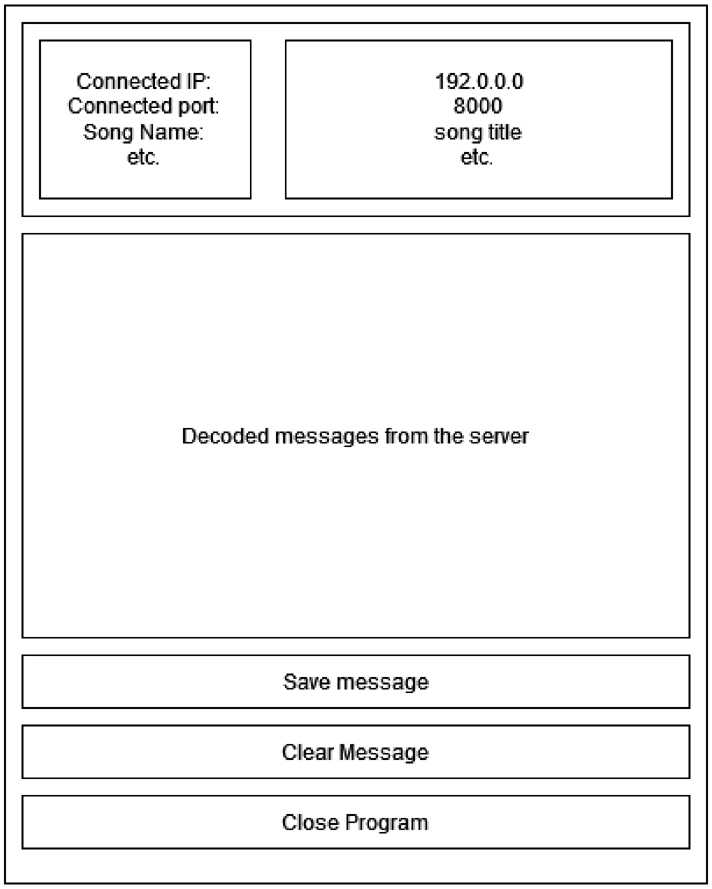
```
Extractdata(data):
    for I in data:
        scan the array for the key value
        if the key value is found convert it back to an ascii character and add it to the extracted
values in the GUI
```

The extract data function works by scanning the incoming data for the key value hidden in the array of data. When it finds a value that matches the key the value from the first audio channel is extracted and sent to the GUI to be shown. This function runs on a secondary thread to reduce the amount of impact it has on the audio stream and the packet processing.

2.5.3 Wire Frames

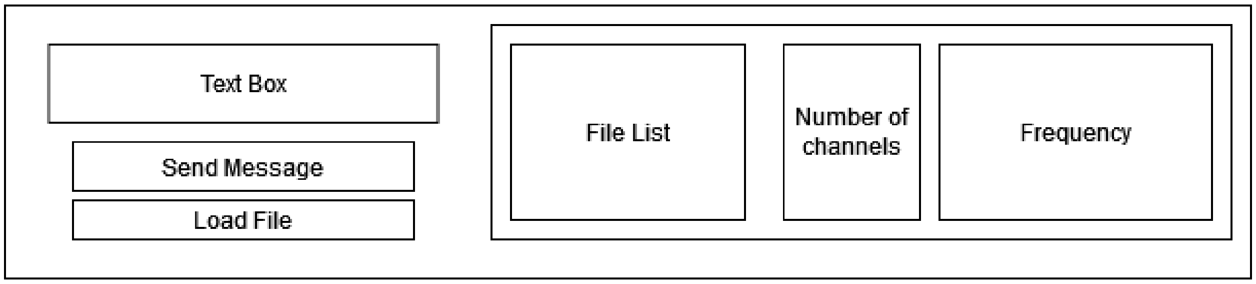
The idea I had for the UI was to have a basic UI that would show the incoming data for the client and some basic information about the streaming music. The buttons would have actions tied to them to either save the current decoded messages, clear the messages, and close the application. Upon decoding a message, the info is displayed in the message box above the buttons and is visible to the user.

Client



The client GUI has 3 major parts to it. The first is the connection information that is displayed at the top of the window. The second part is the text window in the middle of the window. This window is not editable by the user but shows the decoded messages from the server to the client devices. This window is automatically filled and cleared when a message is saved to a file or cleared using the buttons below. The final part of the GUI is the three buttons on the bottom. The first will take the contents of the window and save them to a file. The second will clear the message box but will not save the contents and the last will close the connection with the server and close the application. This function is also ran when the user closes the window.

Server



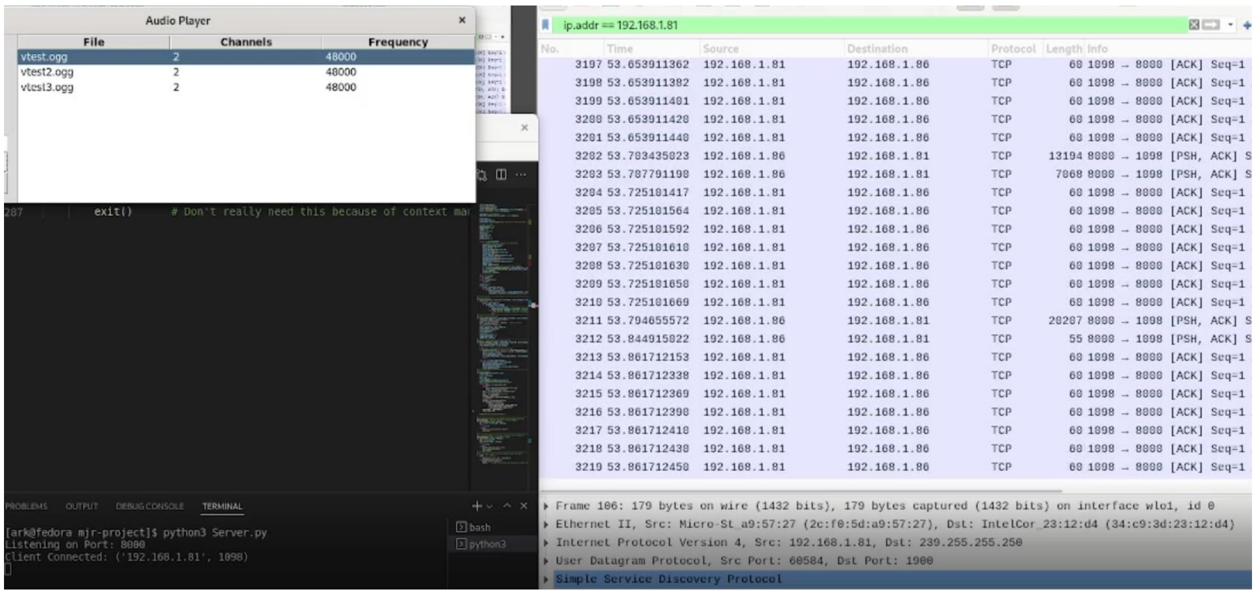
The Server has a basic GUI that allows the user to see the available songs that they can play. The song can be changed by having double clicking the file on the list. The text box was originally designed for debugging but also allowed for quick messages to be sent for the server device. The send message button clears the input text box and adds the values to the steganography buffer. And the load file button creates a file search that the user can click a file to load its contents to the buffer instead.

2.6. Testing Details and Results

To test this application, I used a Pass/Fail metric that I originally created with the proposal for this project. For each part of the metric I conducted tests to ensure that the application is able to complete them.

2.6.1 Audio Stream Testing

Case 1	Server Is able to connect to client and send traffic
Verification	Ensure that server sockets and client application is properly configured
Pass Condition	The client device receives packets from server
Fail Condition	Client device does not receive packets from server


Audio Player

File	Channels	Frequency
vtest.ogg	2	48000
vtest2.ogg	2	48000
vtest3.ogg	2	48000

```
        exit()      # Don't really need this because of context ma
```

```
[ark@fedora mjr-project]$ python3 Server.py
Listening on Port: 8000
Client Connected: ('192.168.1.81', 1098)
```

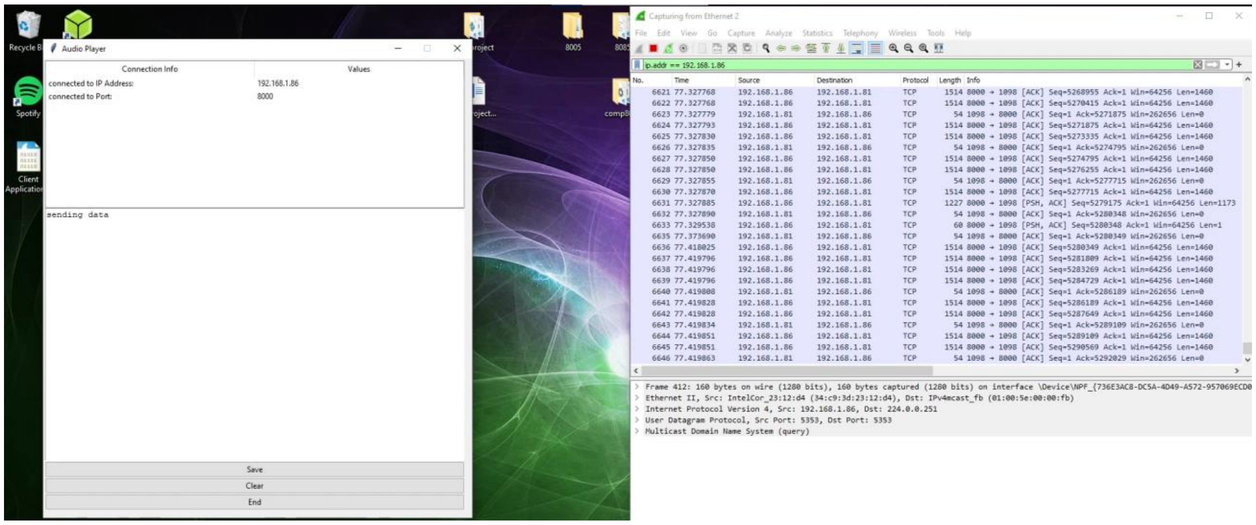
```
ip.addr == 192.168.1.81
No.   Time          Source        Destination   Protocol Length Info
 3197 53.653911362  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3198 53.653911382  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3199 53.653911401  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3200 53.653911420  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3201 53.653911440  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3202 53.703435023  192.168.1.86  192.168.1.81  TCP      13194 8000 → 1098 [PSH, ACK] S
 3203 53.707791198  192.168.1.86  192.168.1.81  TCP       7068 8000 → 1098 [PSH, ACK] S
 3204 53.725101417  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3205 53.725101564  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3206 53.725101592  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3207 53.725101610  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3208 53.725101630  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3209 53.725101650  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3210 53.725101669  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3211 53.794655572  192.168.1.86  192.168.1.81  TCP      20207 8000 → 1098 [PSH, ACK] S
 3212 53.844915022  192.168.1.86  192.168.1.81  TCP         55 8000 → 1098 [PSH, ACK] S
 3213 53.861712153  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3214 53.861712338  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3215 53.861712369  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3216 53.861712390  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3217 53.861712410  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3218 53.861712430  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1
 3219 53.861712450  192.168.1.81  192.168.1.86  TCP         60 1098 → 8000 [ACK] Seq=1

▸ Frame 106: 179 bytes on wire (1432 bits), 179 bytes captured (1432 bits) on interface wlo1, id 0
▸ Ethernet II, Src: Micro-St_a9:57:27 (2c:f0:5d:a9:57:27), Dst: IntelCor_23:12:d4 (34:c9:3d:23:12:d4)
▸ Internet Protocol Version 4, Src: 192.168.1.81, Dst: 239.255.255.250
▸ User Datagram Protocol, Src Port: 60584, Dst Port: 1900
▸ Simple Service Discovery Protocol
```

Shown in the playback that a user can connect to the server and traffic is being sent between devices. This case passes because the server can accept connections from the client devices and then send data to them. The above photo shows the output on the server along with the packet capture of when a user is connected to the server.

Case 2	Client Device can play audio from server stream
Verification	Ensure implementation of audio decoding and playing is functional
Pass Condition	Client Device can play audio that is being sent from the server
Fail Condition	Client device is unable to play audio

Upon testing, once a client is connected to the server it will attempt to automatically play the data that it receives. This data played using python sound devices utilizing the outputstream method.


Recycle B
Spotify
Client Applicatio
Audio Player
Save
Clear
End
Capturing from Ethernet 2
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Connection Info	Values
connected to IP Address:	192.168.1.86
connected to Port:	8000

```
sending data
```

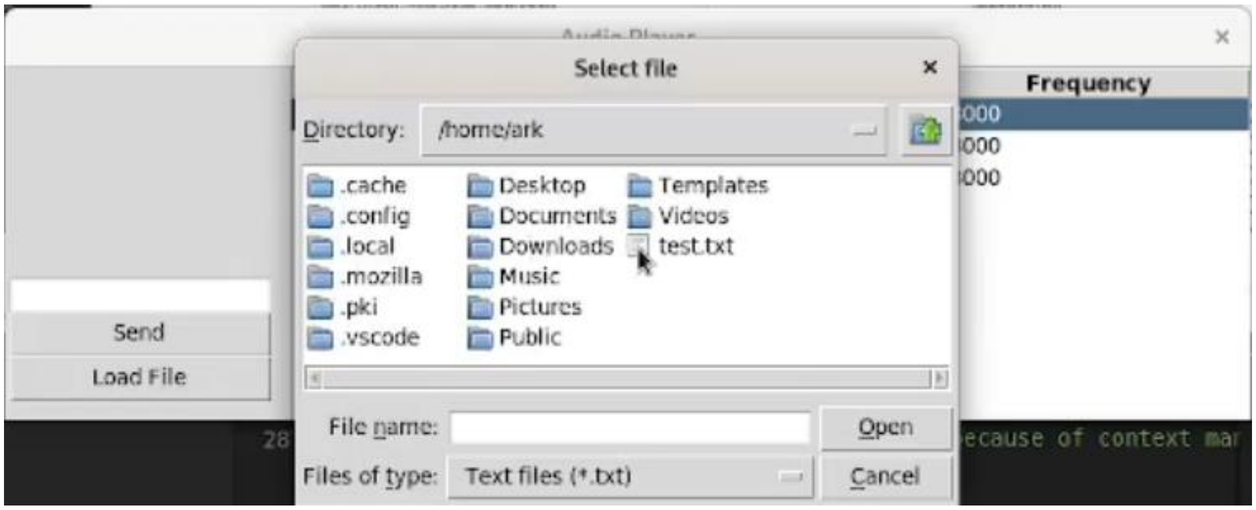
```
ip.addr == 192.168.1.86
No.   Time       Source        Destination   Protocol Length Info
 6621 77.327768  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5268955 Ack=1 Win=64256 Len=1460
 6622 77.327768  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5270415 Ack=1 Win=64256 Len=1460
 6623 77.327779  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5271875 Win=262656 Len=0
 6624 77.327793  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5271875 Ack=1 Win=64256 Len=1460
 6625 77.327830  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5273335 Ack=1 Win=64256 Len=1460
 6626 77.327835  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5274795 Win=262656 Len=0
 6627 77.327850  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5274795 Ack=1 Win=64256 Len=1460
 6628 77.327850  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5276255 Ack=1 Win=64256 Len=1460
 6629 77.327855  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5277715 Win=262656 Len=0
 6630 77.327870  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5277715 Ack=1 Win=64256 Len=1460
 6631 77.327885  192.168.1.86  192.168.1.81  TCP      1227 8000 → 1098 [PSH, ACK] Seq=5279175 Ack=1 Win=64256 Len=1173
 6632 77.327890  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5280348 Win=262656 Len=0
 6633 77.329538  192.168.1.86  192.168.1.81  TCP        60 8000 → 1098 [PSH, ACK] Seq=5280348 Ack=1 Win=64256 Len=1
 6635 77.373690  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5280349 Win=262656 Len=0
 6636 77.418025  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5280349 Ack=1 Win=64256 Len=1460
 6637 77.419796  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5281809 Ack=1 Win=64256 Len=1460
 6638 77.419796  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5283269 Ack=1 Win=64256 Len=1460
 6639 77.419796  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5284729 Ack=1 Win=64256 Len=1460
 6640 77.419808  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5286189 Win=262656 Len=0
 6641 77.419828  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5286189 Ack=1 Win=64256 Len=1460
 6642 77.419828  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5287649 Ack=1 Win=64256 Len=1460
 6643 77.419834  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5289109 Win=262656 Len=0
 6644 77.419851  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5289109 Ack=1 Win=64256 Len=1460
 6645 77.419851  192.168.1.86  192.168.1.81  TCP      1514 8000 → 1098 [ACK] Seq=5290569 Ack=1 Win=64256 Len=1460
 6646 77.419863  192.168.1.81  192.168.1.86  TCP        54 1098 → 8000 [ACK] Seq=1 Ack=5292029 Win=262656 Len=0

> Frame 412: 160 bytes on wire (1280 bits), 160 bytes captured (1280 bits) on interface \Device\NPF_{736E3AC8-DC5A-4D49-A572-957069ECD0
> Ethernet II, Src: IntelCor_23:12:d4 (34:c9:3d:23:12:d4), Dst: IPv4mcast_fb (01:00:5e:00:00:fb)
> Internet Protocol Version 4, Src: 192.168.1.86, Dst: 224.0.0.251
> User Datagram Protocol, Src Port: 5353, Dst Port: 5353
> Multicast Domain Name System (query)
```

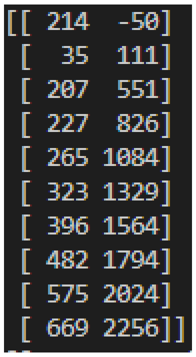
2.6.2 Steganography Testing

Case 1	Server can encode a message into packets as they are being sent
Verification	Ensure implementation of the encoder in the server is functional with live encoding and file encoding
Pass Condition	Packets that are being sent from the server have a portion modified with the encoded data
Fail Condition	Server is not able to encode packets and send them

The server uses two ways to gain input from the user and then encodes parts of the message into the audio data. It will ether take a direct input from the user via the input box or the user can load a text file to send using the load file button. This will add the message to a buffer that is being merged with the audio data. The data is then encoded into the audio data using a key system to be relocated and decoded.



The above photo shows the ways that the user can input data to be encoded into the audio data. The user can use the textbox above the Send button to add to the buffer or use the Load File button to overwrite the current buffer and start sending a text file instead.

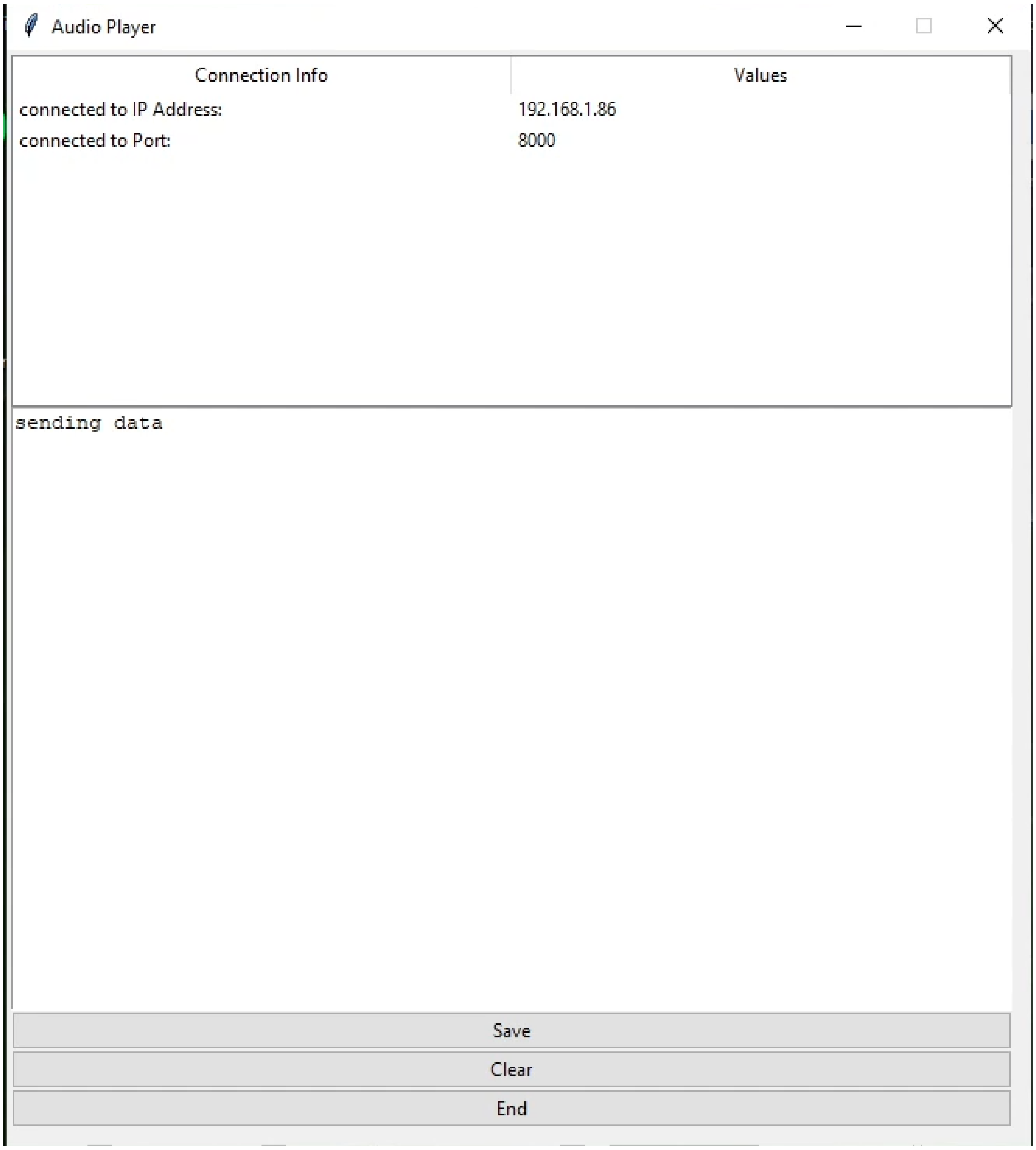


```
[[ 214  -50]
 [  35  111]
 [ 207  551]
 [ 227  826]
 [ 265 1084]
 [ 323 1329]
 [ 396 1564]
 [ 482 1794]
 [ 575 2024]
 [ 669 2256]]
```

The above snippet shows the first 10 entries in the audio data array. The data is encoded with the key 111 for retrieval.

Case 2	Client can decode message from audio stream
Verification	Ensure implementation of audio decoder can separate the message from the audio data and attach it to the rest of the message being sent
Pass Condition	Client Device can extract message from audio stream being sent from server
Fail Condition	Client device is unable to extract message from audio stream

Using the key, the audio stream is then able to extract and decode the encoded data. The data is then shown on the GUI to the user. The client scans the incoming data for the key and will then extract the data where the key is present. The issue that arises from this is that if the key is a common number in audio, then more noise or accidentals will appear from the in the output.



Connection Info	Values
connected to IP Address:	192.168.1.86
connected to Port:	8000

```
sending data
```

The input being sent to the client was a text file with a bunch of random inputs attached to it to show the client being able to extract characters from the stream. The string “sending data” was inputted on the server device to be encoded to the client devices.

2.6.3 Live Testing

Case 1	Listener is unable to detect a change in the audio quality as the audio stream is playing
Verification	Ensure that the amount of encoded data does not interrupt the listener and expose the modified packets
Pass Condition	Listener is unable to detect change in audio
Fail Condition	Listener can detect change in audio

The live testing showed many different results. For this project I had a test group of 12 people ranging from family and friends. The results were quite mixed as some people were able to spot where the data was modified, and some were unable to find it. This puzzled me when looking at the results until I noticed a similarity in the environment of the users. Users with higher quality speakers or headphones were able to hear the modified audio while users who were using lower quality speakers such as the ones included with a monitor were unable to detect the changes.

	Sound device	Detected?
Tester 1	Razor headset	Yes
Tester 2	Sony Earbuds	No
Tester 3	Monitor speaker	No
Tester 4	Monitor speaker	No
Tester 5	Corsair Headset	No
Tester 6	Beats Headset	Yes
Tester 7	Sony Headset	Yes
Tester 8	Bose headphones	No
Tester 9	Monitor speaker	No
Tester10	Skullcandy headset	Yes
Tester 11	Corsair Headset	No
Tester 12	Apple headphones	No

This shows that the use of a key to extract the hidden message from the audio data while having an effect of the audio wave is only detectable on higher end devices but on a low-end speaker or headset would be undetectable.

For this testing I also wanted to see how the effects of using different genres of music to hide the information and which genre would be more effective. For this test I used the same tester on their devices and played music of multiple genres ranging from dance, classical, pop, rock. For this I found that music that is more dynamic and fast had a lower detectability then then music that was slower and more straight forward.

Case 2	Client Device can play audio from server stream
Verification	Ensure implementation of audio decoding and playing is functional
Pass Condition	Client Device can play audio that is being sent from the server
Fail Condition	Client device is unable to play audio

When using the client application once it connects to the server it will attempt to automatically start playing the data that is receives from the server. This data once processed is sent to a buffer that is then played.

2.6.4 Scalability Testing

Case 1	Server can handle multiple connections (50 listeners) simultaneously
Verification	Ensure implementation of sockets can accommodate multiple connections simultaneously.
Pass Condition	Server can stream audio to each device connected without slowing connection or crashing
Fail Condition	Server is unable to handle multiple connections and stream fails or crashes

In testing I used two computers acting as clients both running 25 clients simultaneously. During the testing the server was able to handle the clients with a small impact to performance. This impact was handled by prebuffering on the client devices to account for the server handling many requests. After 100 clients are connected the servers performance becomes noticeably worse as the stream starts to stutter with the high traffic.

2.7. Implications of Implementation

I used an Agile methodology to manage the process of this project. I attempted to break the project into small issues that I found and would attempt to keep the base project able to be functional as I continued to work on it.

The creation of the application was split into 3 parts: the audio stream, the steganography encoder, and the GUI creation.

The audio stream first needed a server to be ran on. To create the server I decided to use the Epoll API even though it limited my server to be running on a linux operating system. I chose this as I was recently learning about the API and found that it would potentially be the best one for handling the simultaneous requests. Next I originally attempted to use a WAV file for the audio but quickly found that the files were too big to be streamed and the requirements to play them would be practically impossible. After doing some research into other streaming services I found that the file type commonly used are Vorpus encoded OGG files. These retain a high quality of the original recording but compress the data into a much smaller file allowing for streaming services to be able to keep up with the playback. After deciding this I needed to determine how the client would be able to play the data. For this I used python sound devices which allows for the playback with an input resembling a nparray. Upon my first prototype I found that the playback would still stutter even with the new file type and had to determine the problem. I found that the system needed a buffer of preprocessed data that would then be able to steadily feed the stream to keep the playback consistent.

The next step was the creation of the steganography encoder. For this I decided to use a key system to hide the data and then retrieve it on the client device. This system would take an input from the user and hide it in a portion of the audio data. Then on the client device the data would be scanned for the preset key and the bits would be extracted and decoded.

The final step was the creation of the GUI. For this I decided to use TKinter to create the GUI as it was a library I had used previously and attempted to create a basic GUI that would be able to take an input from the user on the server and then display the decoded message on the client. Then from this I added some basic functionality such as saving the decoded messages and changing the currently playing song.

2.8. Innovation

The innovation in this project comes from the environment that the application is in. The use of steganography in a Realtime environment is to my research unheard of. The normal type of steganography is that the host file and the hidden message is merged and then the entire file is then sent to a client. This project instead is doing the action of merging the file and hidden message together in Realtime.

This project also makes the interception of a message harder for a third party as the audio stream is constantly sending data out and it would be impossible to detect which packets have parts of the message encoded into them without needing to extract a very large amount of data.

The way I encoded the data is utilizing the structure of the audio data. The format of stereo audio is two channels that can be seen as an array of two values. Using this format I set a key for the data and encode each part of the data into a number to be also disguised as. The client would know this key that the encoded bits are being sent on and would extract the encoded bit at the location where the key is also present.

In the snippet below the key that data is being encoded is 111 and upon finding the encoded section the data would take the value in the other channel (97) and convert it back into an ascii character.

```
[[ 196, 174] [   97, 111] [ 163, 187] [ 105, 186] [ 174, 176] [ 182, 176] [ 162, 128]]
```

The usage of a key and retrieval system makes the encoded bits harder to locate as they look like normal parts of the audio data. The extraction is also light weight as the client device only needs to look to find a single key in the data and extract the bit from it. This extraction method needs to be light weight as if it takes too long then the audio stream may start to stutter as the client can't keep up with the requirement of 48000 bits per second to remain stable playing.

The base steganography works by taking in two separate buffers and then combining them to create the data that is being sent to each client device. The first buffer consists of the audio data mentioned above. The second buffer consists of the sting of data being taken from the GUI. This second buffer is an array of numbers that represent ascii characters when the buffers are spliced together the data from the second string is replacing the data in the first channel while the second channel is changed to the retrieval key for the client device to extract.

2.9. Complexity

The complexity of this project I found came a lot from the need for design and being able to keep up with the real-time environment. The goal I set was to be able to have at least 50 simultaneous connections. This originally when I wrote it seemed like a very easy task. At the time I didn't know much about how streaming worked but quickly found how much data needs to be transferred to be able to keep up with a livestream.

For this system to work it would need to be able to send out 48000 bits to each client per second, with no errors to have a steady playback for the user. The way that I attempt to keep the playback steady is using a playback buffer in which the client will receive data ahead of the current location in the song and will preprocess the data to be fed into the stream. This allows the stream to remain steady incase the connection becomes unsteady.

2.10. Technologies

Some of the technologies and libraries that I used for my project are:

- Python 3
- Tkinter (GUI)
- Python sounddevice (playing audio)
- Numpy
- Pyogg (.ogg file handling)
- Github

2.11. Future Enhancements

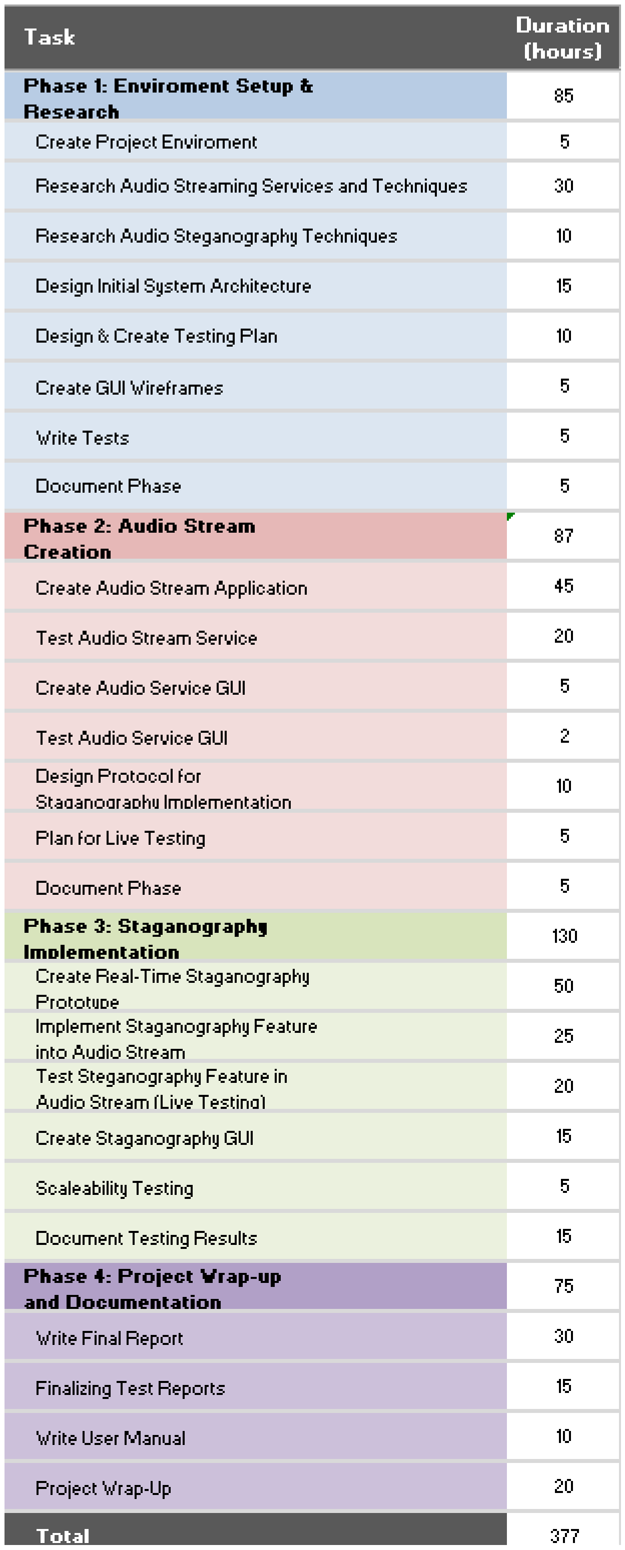
One of the major enhancements I would want to eventually implement would be a way to dynamically assign where the hidden information would be. It would have to pre-scan the data and find the best fit for the parts of the message to reside in so more data could be encoded while not increasing the vulnerability of being detected by the listener.

Another enhancement I wanted to originally implement was showing the modified sound wave versus the original soundwave. This could be done using matpyplot but in my attempts the making of the graph would overload the server and caused it to crash and not meet the scalability requirements.

2.12. Timeline and Milestones

I originally separated the project into 4 phases the research phase, audio stream phase, steganography phase, and then testing and project wrap up phase. This schedule is what I attempted to stick to with using the milestones listed in the gantt chart to see how far along in the project I was.

I followed an agile approach to creating this application using multiple iterations of each service and attempting to improve upon each part while working. This led to times when I would be having to determine if a newer way would be better then the original implementation.



Task	Duration (hours)
Phase 1: Enviroment Setup & Research	85
Create Project Enviroment	5
Research Audio Streaming Services and Techniques	30
Research Audio Steganography Techniques	10
Design Initial System Architecture	15
Design & Create Testing Plan	10
Create GUI Wireframes	5
Write Tests	5
Document Phase	5
Phase 2: Audio Stream Creation	87
Create Audio Stream Application	45
Test Audio Stream Service	20
Create Audio Service GUI	5
Test Audio Service GUI	2
Design Protocol for Staganography Implementation	10
Plan for Live Testing	5
Document Phase	5
Phase 3: Staganography Implementation	130
Create Real-Time Staganography Prototype	50
Implement Staganography Feature into Audio Stream	25
Test Steganography Feature in Audio Stream (Live Testing)	20
Create Staganography GUI	15
Scaleability Testing	5
Document Testing Results	15
Phase 4: Project Wrap-up and Documentation	75
Write Final Report	30
Finalizing Test Reports	15
Write User Manual	10
Project Wrap-Up	20
Total	377

3. Conclusion

This project helped me improve my understanding of data communications and designing for scalability. This project is one of the first times I have used TKinter and created a working GUI for an application. This project reminded me to use the skills I learned from the project management classes I took during my diploma and helped organize myself for this project.

3.1. Lessons Learned

During this project I learned to research more into the technologies that I am deciding to use more before attempting to use them. I found that some of the libraries I attempted to use were deprecated or no longer updated. This came when I originally planned to use a library called pyaudio. This library looked like it would cover all that I was trying to do for the handling the audio part of this application, but I found when trying to use it that it was created for python 2.7 not 3 and was unusable with my environment.

I also learned a lot about how buffering works in streaming. This was always something that I was curious about as I tend to listen to music quite often and was wondering what was happening when an application would say that it is buffering before playback.

3.2. Closing Remarks

With this project I entered wanting to learn and grow in my understanding of how networking and data communications work. Through this project I was able to further my understanding and test my ability to design a server that was able to handle a high workload and preform a complex task at the same time. This project also allowed me to test and understand how to preform stress testing along with user testing which I have never done before.

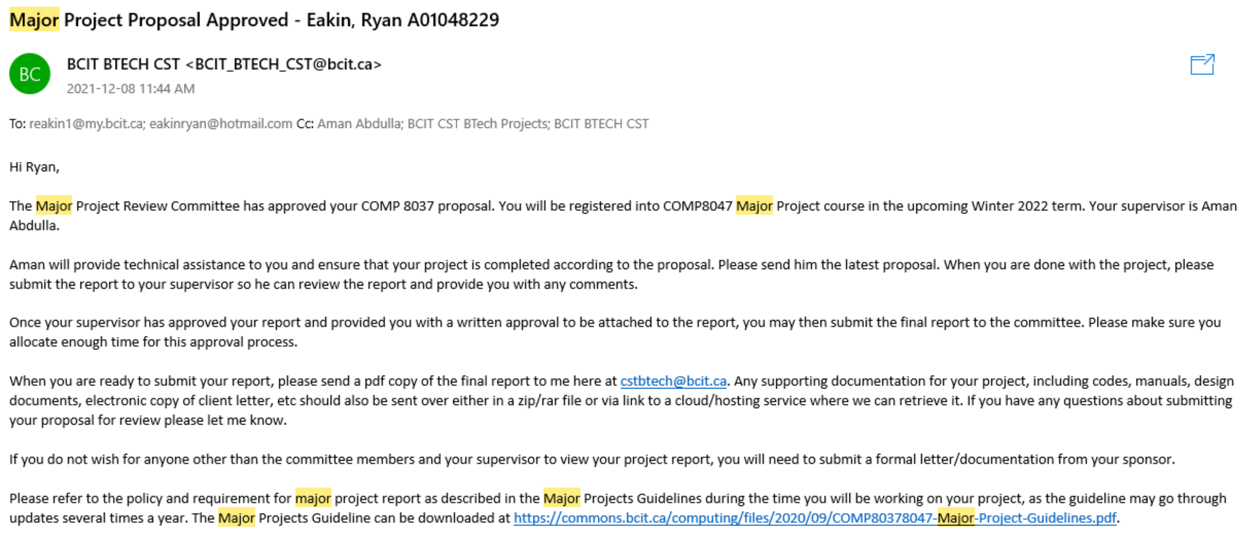
Overall, even though there are parts of this project that I want and plan to further improve on I am happy in my growth with this project and am grateful for the opportunity to work on this.

4. Appendix

4.1. Approved Proposal

See attached file.

4.2. Project Supervisor Approvals



Major Project Proposal Approved - Eakin, Ryan A01048229

BC

BCIT BTECH CST <BCIT_BTECH_CST@bcit.ca>

2021-12-08 11:44 AM

To: reakin1@my.bcit.ca; eakinryan@hotmail.com Cc: Aman Abdulla; BCIT CST BTech Projects; BCIT BTECH CST

Hi Ryan,

The Major Project Review Committee has approved your COMP 8037 proposal. You will be registered into COMP8047 Major Project course in the upcoming Winter 2022 term. Your supervisor is Aman Abdulla.

Aman will provide technical assistance to you and ensure that your project is completed according to the proposal. Please send him the latest proposal. When you are done with the project, please submit the report to your supervisor so he can review the report and provide you with any comments.

Once your supervisor has approved your report and provided you with a written approval to be attached to the report, you may then submit the final report to the committee. Please make sure you allocate enough time for this approval process.

When you are ready to submit your report, please send a pdf copy of the final report to me here at cstbtech@bcit.ca. Any supporting documentation for your project, including codes, manuals, design documents, electronic copy of client letter, etc should also be sent over either in a zip/rar file or via link to a cloud/hosting service where we can retrieve it. If you have any questions about submitting your proposal for review please let me know.

If you do not wish for anyone other than the committee members and your supervisor to view your project report, you will need to submit a formal letter/documentation from your sponsor.

Please refer to the policy and requirement for major project report as described in the Major Projects Guidelines during the time you will be working on your project, as the guideline may go through updates several times a year. The Major Projects Guideline can be downloaded at https://commons.bcit.ca/computing/files/2020/09/COMP80378047-Major-Project-Guidelines.pdf.

5. References

- COMP 8005 (notes)
- Python Sounddevices https://python-sounddevice.readthedocs.io/en/0.4.4/
- Pyogg https://pyogg.readthedocs.io/en/latest/
- Numpy https://numpy.org/doc/stable/
- TKinter https://docs.python.org/3/library/tkinter.html

6. Change Log

May 30, 2022 – Expanded 2.1 Background to include more information on the background description of the steganography problem.
May 30, 2022 – Added Psudo code section 2.5.2 to further explain how the steganography encoding works.