For my final project I decided to implement a client-server audio and text chat app in go. To begin the project, I needed to determine if it were possible to play audio in go. In research that I discovered this [tutorial on medium](https://medium.com/@valentijnnieman_79984/how-to-build-an-audio-streaming-server-in-go-part-1-1676eed93021) that was trying to accomplish a similar goal. They implemented the portaudio library via the [go api here](github.com/gordonklaus/portaudio). This library allowed the code to read samples from the microphone and then play them back on the client side. To build this library on Linux the portaudio19-dev is needed and on Windows the steps [here](https://github.com/PortAudio/portaudio/wiki/Notes_about_building_PortAudio_with_MinGW) need to be followed, after the cloning the git repo <https://github.com/PortAudio/portaudio>, the specific commands I used while in a MinGW64 were ./configure --with-winapi=wmme; make; make install; then whenever I built or ran the go application, I would execute the build command in the MinGW64 environment. The generated executable can then be used outside of that environment as long as the libportaudio-2.dll library file is somewhere on the path or in the same directory. I included compiled executables and the library in the github repo so that it is easier to just run the program to test it. The app itself is set up so that the server can communicate via audio to each client and the clients can communicate back to the server via text messages. The audio is one way from the server to the client in a radio style. The amount of audio to send at a time is determined by the value entered by the user on the server side. This value must be synchronized with the client or the audio will either repeat or cut off. This synchronization is done via the /bufsize endpoint. Clients connected to the /audio server path will receive audio until they stop the program by entering the /stop command. Clients can communicate to the server by entering any text (besides /stop and /name{anything}) and the server will display this text along with the name of the client that sent the data. This value defaults to the clients ip address but can be change by entering /name{newnamehere} (no space between /name and the new name). I attempted to have the clients be able to also read text messages sent to the server but could not get the right response, using gRPC would likely be the better way to do this and not the simple POST and GET method that I used. I think this method could also be used to enable two-way audio communication, but I am not sure about that and the websocket go library may be a better way to implement that. One of the other strange behaviors that I noticed is that the fmt.Scanln is acting like fmt.Scan and reading up until the first space instead of reading until the newline value that the [documentation](https://pkg.go.dev/fmt#Scanln) says it should read until. The delay in the audio is roughly equal to the time to send data from server to client (calculated as the time it takes /bufsize to return a value) plus two times the length of the buffer in seconds (because the server takes that amount of time to read from the microphone and then the client takes that amount of time to send the audio to be played). There is also likely another aspect of delay in the time it takes the c++ api to respond to the request, but I could not measure this extra time and it is likely very small. For a way in that I think that this application could be used I can see a system with one main controller who can speak to a number of people that need to be directed or managed. The audio would play over a speaker on the clients devices so that they do not miss an instruction and then if they need to communicate their status back they can do so via text so as not to have a lot of people speaking at once. This could also be something like a large conference where one person were speaking and viewers could ask private questions in the chat to the presenter so they can address them without being interrupted.