Comp 112 Final Project
Music Server/Client Program
Shaylynn Reilly and Seth Kahn
Git Repo: https://github.com/kahns729/comp112-final-project

Overview:

Our project is a music streaming server/client pair, using python3, pyaudio, and pydub (see README for details). Our project is broken down into 3 major design elements: a music stream, a requests channel, and peer to peer connectivity.

Design:

1. Stream

We designed our stream to use PyAudio and PyDub, and to run on one port. The server reads in a .mp3 file from a directory of songs and breaks this into “chunks”, which it streams to a client. A directory in a level above the directory the server is running from called “songs” must exist (../songs) in order for the program to run. If there have been no songs requested, the server chooses a song randomly from our songs directory to play, but otherwise plays songs in the order that they were requested by clients.

2. Requests

We use a second port to handle requests to the server for the list of songs available, to request a song to play, and for a list of the songs that have been requested. Requests are then sent from the client directly to the server and the server will send back an appropriate response.

Syntax for 3 types of requests: SONGS, PLAY [song_name]/PLAY [song_index], REQUESTS

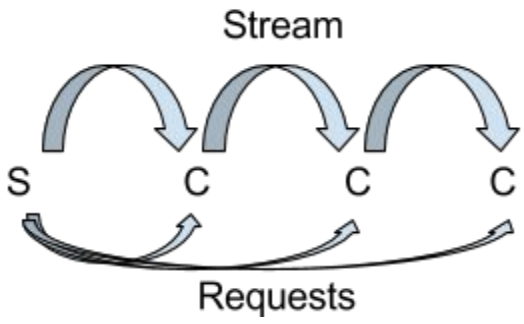
A response to SONGS will give a numbered list of the .mp3 files available for streaming in the songs directory

A response to PLAY will either state that “Song [song_name] has been requested!”, “Song has already been requested” if the song is already in the list of requested songs, or “Song does not exist” if the index is out of range or the song name does not exist as an mp3 file in the songs directory.

A response to REQUESTS will give a numbered list of the .mp3 files that have been requested, with the 1st as the next song to be played

3. Peer to peer

The peer to peer element of our project is designed similarly to a linked list. The peer to peer aspect only applies to the streaming part of the application, whereas all requests go directly from each client to the server. The first client to connect will connect directly to the server on both ports, with one port receiving a music stream, and the other handling requests. The second-nth clients will then connect to the last client in our “client list” on one port, and to the server on another to handle requests.



Implementation Overview:

We implemented the application using Python 3. We use two Python modules: one to represent the server functionality, and another to represent the client/peer functionality. For every client, we maintain a persistent *request* TCP connection between the client and the server. This socket will handle any user requests for information from the server.

For each client, there is also a *streaming* TCP connection. This connection handles the transmission of music between clients and from the server. This stream is constantly busy transporting chunks of music data. The first such stream is maintained between the server and the first client to connect. The second stream is open when a second client connects, and it transmits data between the first client and second client (peer to peer). This pattern continues so that every node in the chain (which starts with the server) is serving music data to the next node.

Implementation of **stream.py** (the server module):

Our server runs infinitely, until it is killed by whoever maintains it. We use multiple threads in the server. One thread loops, always looking to accept clients. If a client connects, the server first sends the client information about the current song so that it can play the song back. Then it determines where in the chain of nodes to place them. It does this by choosing either the last client that connected or itself (if no client is connected), and communicating to the newly-connected client the host and port of that node. The client will then connect to the proper node given that information. This thread also establishes the request connection with the client.

The server breaks up the current song file into discrete chunks using **pydub**. It sends each individual chunk to its client, with the “SC” (song chunk) header. Also in the header, we indicate how many bytes the client should expect (the size of the song chunk).

When a song changes, we send the first client an “NS” header, followed by song information about the next song to be played.

If no clients are connected, the server still progresses in the current song at a rate meant to emulate the actual play rate of the song, so that the stream is persistent and always continuing.

The server also has a thread to handle requests from the client. Each unique request (as outlined in the design) is handled by receiving the request, checking the header, and either sending back data (in the case of song list or request list requests), or adding a song to the server’s request queue (if a song is requested). This thread also handles a disconnect request, where a client informs the server that it is no longer streaming. The server removes the client from its list of clients, and streams to a new client if the disconnected client was first in line.

Implementation of **client_stream.py** (the client module):

The client initially connects to the server, whose host and port are provided in the command line arguments. First it receives a data packet containing some information about how to playback the current song. It then receives a message from the server containing the hostname and port of the node that it should connect to. The client then connects to this node, closing the TCP connection previously established between itself and the server.

Also when a client connects, a separate thread established a request TCP connection with the server. This connection handles all SONGS, REQUESTS, and PLAY requests to the server, and handles their responses.

Clients must also be prepared to handle connections from new clients (peers). Similar to the server model, clients have a thread running to receive a client. However, this thread does not loop. The only time peers will be accepted after the first peer is accepted is in the case of a disconnect. When a client disconnects gracefully, it sends information to its client (if it has one) about whom to connect to, and the node serving it will detect the disconnect and start expecting a new connection. Conceptually, this is analogous to deletion from a linked list of peer/server nodes. When a client disconnects, it also informs the server (as described in **stream.py**).