1 Introduction

This document details the design and implementation of a simplified Spotify-like application built using Python. The application enables users to view catalog, manage a playlist of songs by adding, removing, and playing them.

Here's an overview of the key aspects:

- The client application features a menu-driven interface that utilizes a binary tree data structure to navigate available functionalities.
- **Database:** A JSON file named database.json serves as the server's database, storing song information.
- **Client-Server Communication:** The client and server interact through a custom application protocol defined in this document.
- **Logic Distribution:**
    - Client-side logic primarily focuses on navigating the tree menu and triggering actions.
    - In contrast, server-side logic handles the execution of commands received from the client.
- **Code Organization:** The server-side logic resides in server.py and response.py files, while the client-side logic is implemented in client.py and request.py.

2 Client-Side Implementation

2.1 Menu Navigation Using Binary Tree

The main menu of the application is structured as a binary tree for efficient navigation and organization. This tree-based representation allows for a clear and logical arrangement of the available options.

- **Root Node:** The main menu serves as the root node of the binary tree.
- **First Level:** At the first level, the left child node is "Catalog," and the right child node is "Playlist."
- **Playlist Node:** The "Playlist" node further branches into two child nodes: "Design" and "Play."

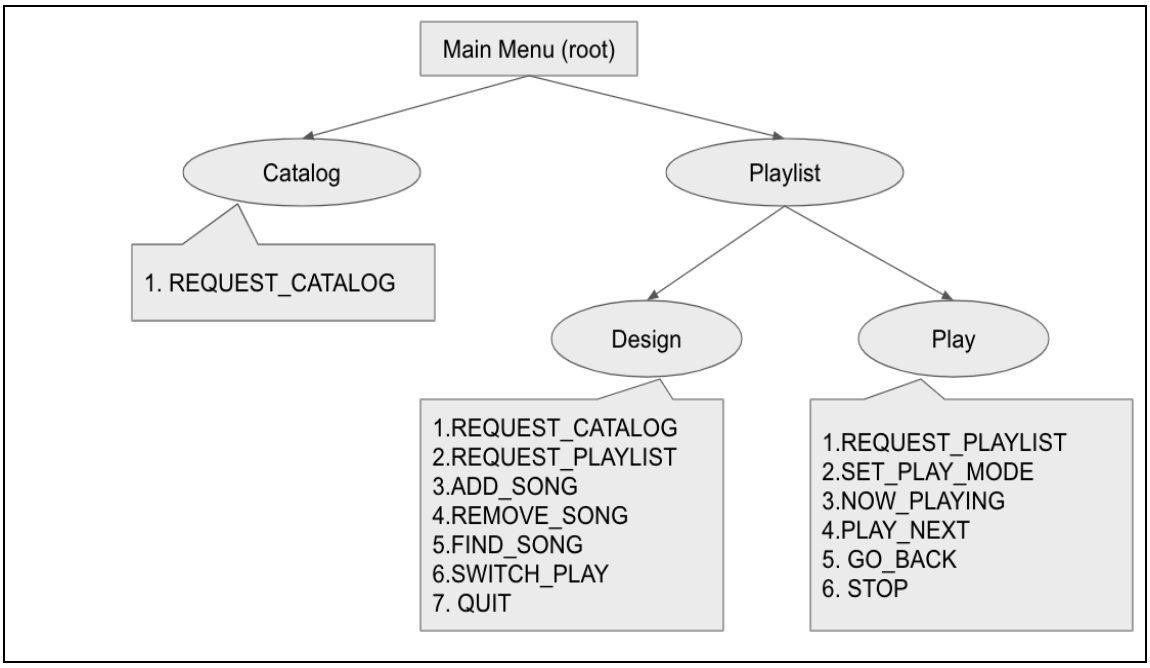

Figure 1: Binary tree Structure of menu navigation.

- **Leaf Nodes:** Each leaf node has an action list. These are the list of commands that can be triggered from the leaf node.

Actions and Commands:

- **Catalog:** The "Catalog" node has a single action, "REQUEST_CATALOG," which retrieves the list of available songs from the catalog.
- **Playlist Design:** This node offers seven actions:
    - "REQUEST_CATALOG": Retrieves the list of available songs from the catalog.
    - "REQUEST_PLAYLIST": Fetches the current playlist information.
    - "ADD_SONG": Adds a specified song to the playlist.
    - "REMOVE_SONG": Removes a specified song from the playlist.
    - "FIND_SONG": Searches for a specific song within the playlist.
    - "SWITCH_PLAY": Transitions from "Design" mode to "Play" mode.
    - "QUIT": Exits the application.
- **Playlist Play:** This node provides six actions:
    - "REQUEST_PLAYLIST": Fetches the current playlist information.
    - "SET_PLAY_MODE": Sets the playback mode (default, shuffle, loop).
    - "NOW_PLAYING": Displays the currently playing song.
    - "PLAY_NEXT": Plays the next song in the playlist.
    - "GO_BACK": Restores the previously played song.
    - "STOP": Stops playback and returns to "Design" mode.

Arguments:

Some commands may require additional arguments, such as the song ID, to be passed when selected. These arguments provide the necessary information for the server to execute the commands correctly.

Code Snippet from client.py:

```
# from client.py
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
   s.connect((HOST, PORT))  # server host and port
   navigate_tree(MENU, s)  # navigates the menu tree
```

`navigate_tree` function contains the logic for interacting with the menu and creating requests that will be passed to the server. The client-side code incorporates a robust error handling mechanism to validate user input. If a user enters invalid commands or arguments, the system will provide appropriate error messages and guidance. Additionally, users can navigate back to the previous menu level at any time by simply entering "0" at the command prompt. This feature offers flexibility and allows users to easily navigate the menu tree.

Note: use appropriate integers on the command line to navigate the menu.

2.2 Request Structure:

Requests sent from the client to the server follow a standardized structure, which is represented as a dictionary. This structure ensures consistent communication and facilitates parsing and processing of requests on the server side.

Code Snippet Of Request Structure:

```
# request.py
# request format that will
command_dict = {
   "header": {
       {
         "user": "Sri Sasmi Polu",
         "GWID": "G40795997",
         "request_time":
datetime.now(timezone.utc).strftime("%y-%m-%d::%H:%M:%S"),
         "client_ip": "127.0.0.1",
         "server_addr": "127.0.0.1:12000"
       }
   },
   "command": command,
```

```
    "args": args
}
```

A request dictionary contains the following key-value pairs:

- **header:** Header calls the get_header() function. This functions gets metadata about the request, including:
    - **user:** The username of the client making the request.
    - **GWID:** the gwid represents the user_id.
    - **timestamp:** The time at which the request was sent in GMT timezone.
    - **client_ip:** The IP address of the client.
    - **server_addr:** The IP address of the server with port number.
- **command:** This field specifies the type of command being sent. This includes all the actions of leaf nodes in the binary tree specified before. Command is a simple string. List of available commands are in section 3.1
- **args:** This field contains any additional arguments required for the specific command as a dictionary. For example, the ADD_SONG command might need the song_ID as an argument. This action specific information is elicited from the user on the client side.

3 Server-Side Implementation:

The server-side implementation is responsible for handling client requests, executing commands, and managing playlist data. This section details the core logic and data structures used within the server to implement the commands.

There are multiple global variables that are used

- **catalog:** This variable stores the entire catalog of songs, including their metadata, which is loaded from database.json.
- **playlist:** Represents the current playlist for the connected user. This playlist is not persisted and is discarded when the user quits the application.
- **og_playlist:** A deep copy of the original playlist, used for restoring the playlist when switched from play to design mode. This playlist is not persisted and is discarded when the user quits the application.

3.1 Command Execution

The server receives client requests in the form of string JSON objects and processes them accordingly. The “handle_client_data” function is responsible for parsing the incoming data, identifying the command, and executing the appropriate logic.

Key Commands and Logic:

- **REQUEST_CATALOG:** Retrieves the entire song catalog from the catalog variable and sends it to the client.
- **REQUEST_PLAYLIST:** Sends the current playlist to the client.
- **ADD_SONG:** Adds the specified song to the playlist.
- **REMOVE_SONG:** Removes the specified song from the playlist.
- **FIND_SONG:** Searches for the specified song in the playlist.
- **SWITCH_PLAY:** Creates a deep copy of the current playlist as og_playlist and transitions to play mode.
- **SET_PLAY_MODE:** Sets the playback mode (default, shuffle, loop).
- **NOW_PLAYING:** Sends the currently playing song from og_playlist to the client.
- **PLAY_NEXT:** Advances the playback to the next song in og_playlist.
- **GO_BACK:** Restores the previously played song in og_playlist.
- **STOP:** Stops playback and transitions back to design mode.

Response Generation: The server returns a response string indicating the command's execution status and output. This response is served to the user on the client side.

4 Client-Server Communication:

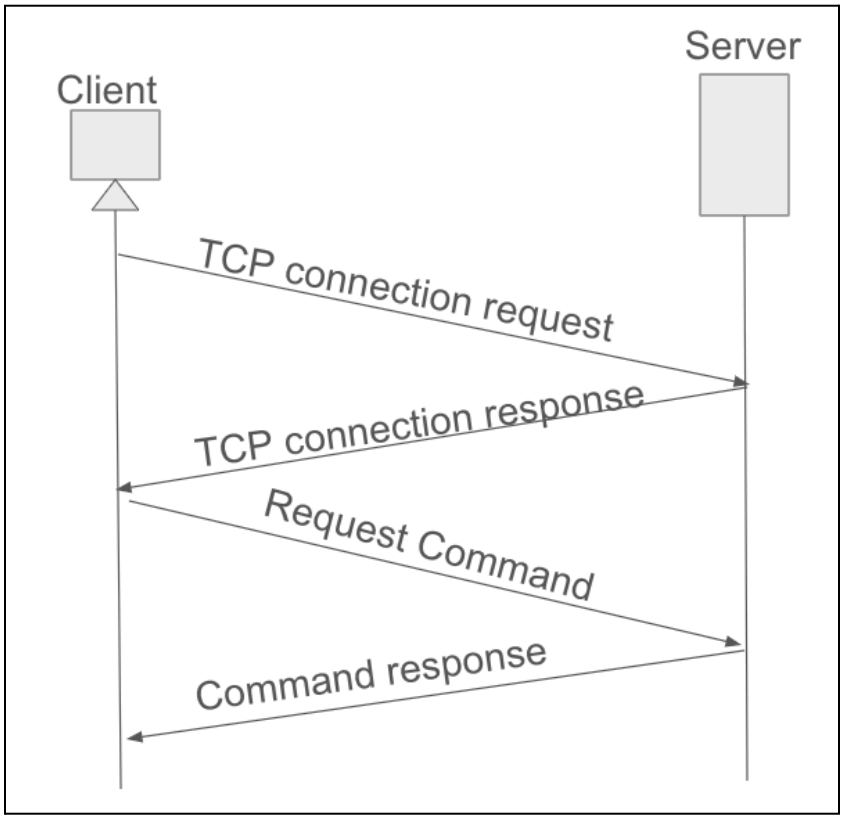

Figure 2: Communication Flow diagram between client and server.

The server uses a TCP socket to listen for incoming connections from clients. When a client connects, the server accepts the connection and enters a loop to continuously receive and

process data. The handle_client_data function is called to process each incoming request, and the response is sent back to the client.

Code Snippet from server.py:

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
   s.bind((HOST, PORT))
   s.listen()
   print(f' Server listening on {HOST}:{PORT}')
   # conn is a client socket object. It is different from s socket
   conn, addr = s.accept()
   with conn:
       while True:
           data = conn.recv(1024).decode()
           if not data:
               break
           print(f"Client address: {addr}")
           # process data send from client
           send_data = handle_client_data(data)
           # send data to client
           conn.send(send_data)
```

Note: The handle_client_data function parses the string JSON, extracts the command and arguments, executes the command, and returns a message (as a string) that will be served on the client side.

Communication between server and client is encoded using the default python encoder.

5 Example Snapshots:

Note: use appropriate integers on the command line to navigate the menu.

Below are the screenshots of both client side and server side when the application is run on the command line interface. Screenshots describe the following scenario.
Client Request: The client requests ‘request_catalog’ in the client CLI.
Server Processing: The server receives the request, processes it, and prepares the catalog data.
Server Response: The server sends the catalog data back to the client.
Client Display: The client receives the catalog data and displays it.

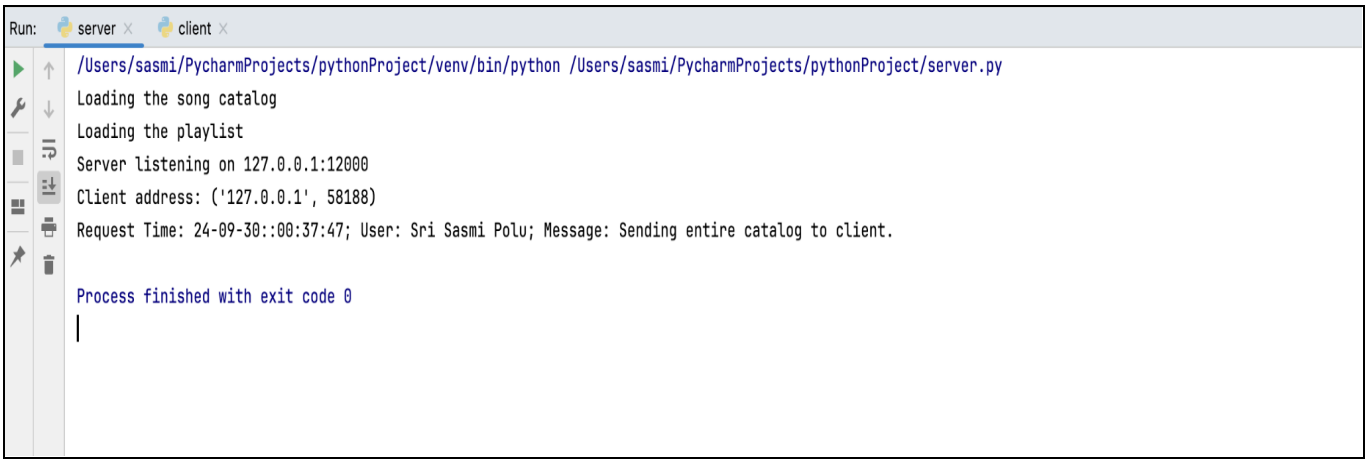

Figure3: Server-Side CLI Screenshot: Processing Request Catalog

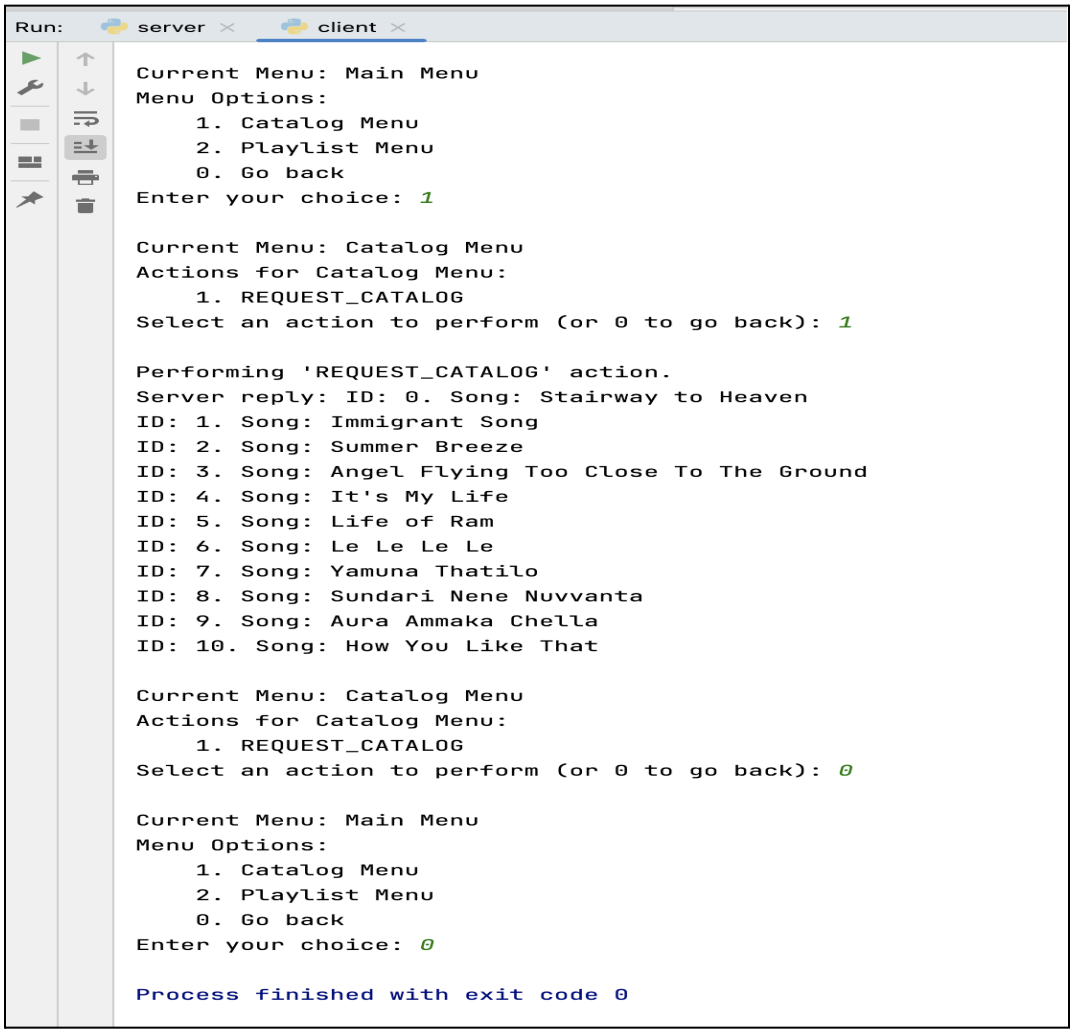

Figure 4: Client-Side CLI Screenshot: Receiving Catalog