Project Report
On

Music Streaming Platform

Submitted by

Akanksha Goel	**T00803**
Rohit Kumar	**T00801**

Under the Guidance of

Vikas Nagpal and Neeraj Arora

Hughes Systique Corporation

Year 2024

Contents

Chapter 1

Introduction

1.1 Problem Statement

The problem involves designing and implementing a multi-threaded client-server music streaming platform *MUSICOPHILE*. The system should facilitate concurrent music streaming from a server to multiple clients. Key functionalities include handling incoming client connections, managing user authentication, serving music content, and supporting playback features like play, pause, resume, and stop. It should prioritize resource efficiency, scalability to handle large user loads, and robustness to manage network disruptions and client disconnections. Ultimately, the objective is to create a reliable and responsive music streaming platform that offers users a seamless and enjoyable music listening experience.

Chapter 2

Literature Review

2.1 Multi-threaded Client server Architecture

A thread is a unit of execution within a process. It runs independently, sharing resources and memory with other threads in the same process. Threads are lightweight and enable concurrent execution, improving system efficiency and performance.

A multi-threaded client-server architecture utilizes multiple threads to handle concurrent client connections in a server application. Server threads listen for incoming client connections and pass them to a pool of worker threads for processing. This architecture improves performance and responsiveness by allowing the server to serve multiple clients simultaneously without blocking the main execution thread.

2.2 SFML: A Multimedia Framework

SFML (Simple and Fast Multimedia Library) is a versatile C++ multimedia framework known for its ease of use and cross-platform compatibility. It offers developers a straightforward API for rendering graphics, playing audio, and managing user input, making it well-suited for a wide range of interactive applications and games.

In the application, SFML is utilized to craft a user-friendly music streaming interface. It features sign-in and sign-out pages, along with a main page showcasing various music genres. Each genre is interactive, allowing users to browse through a curated list of songs and play their preferred tracks. Through skillful integration of SFML's graphical capabilities and event handling mechanisms, the application delivers an intuitive and immersive music streaming experience for users to enjoy.

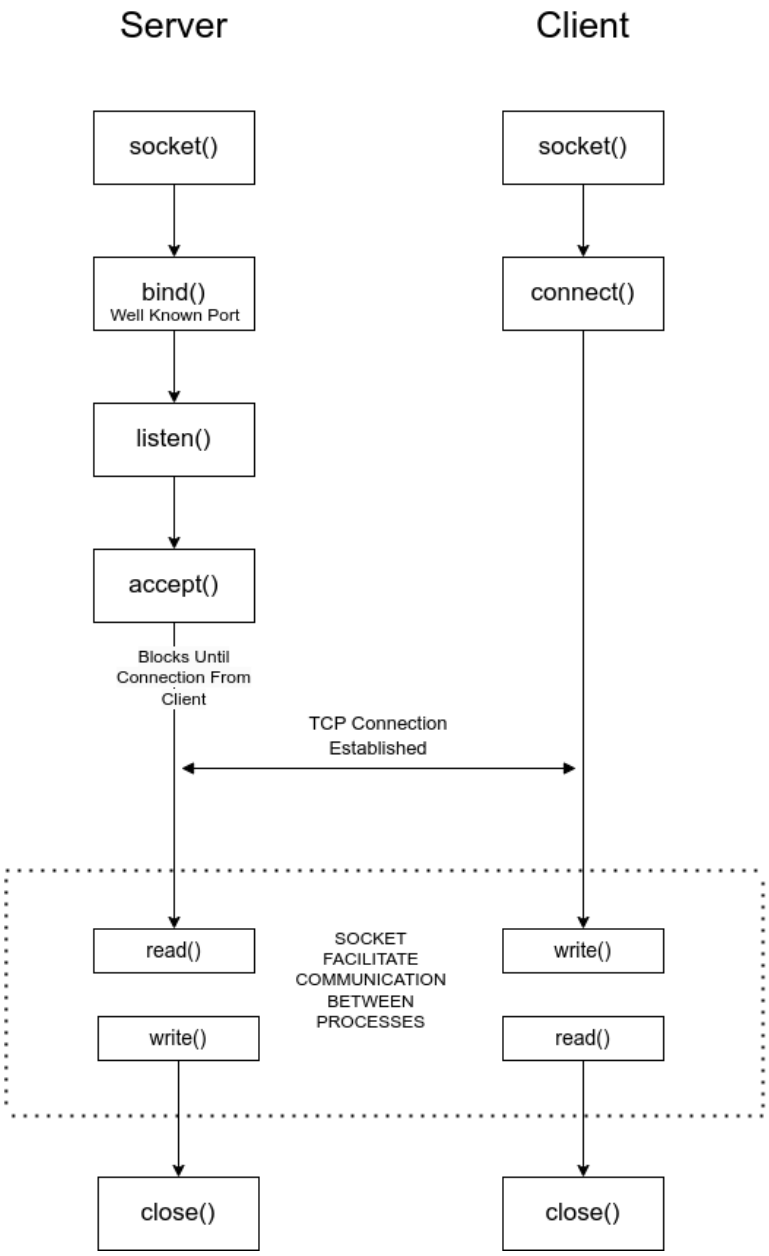
2.3 Irrklang : Audio library

The IrrKlang library provides a powerful audio engine for C++ applications, offering features for playing and managing sound resources. With the ISound* interface, developers can manipulate audio playback instances, control volume, and apply various effects. Among its functions, play2D stands out as a simple yet versatile method for playing 2D sounds. It allows developers to quickly start playing a sound in a 2D audio environment, making it ideal for games, multimedia applications, and interactive experiences.

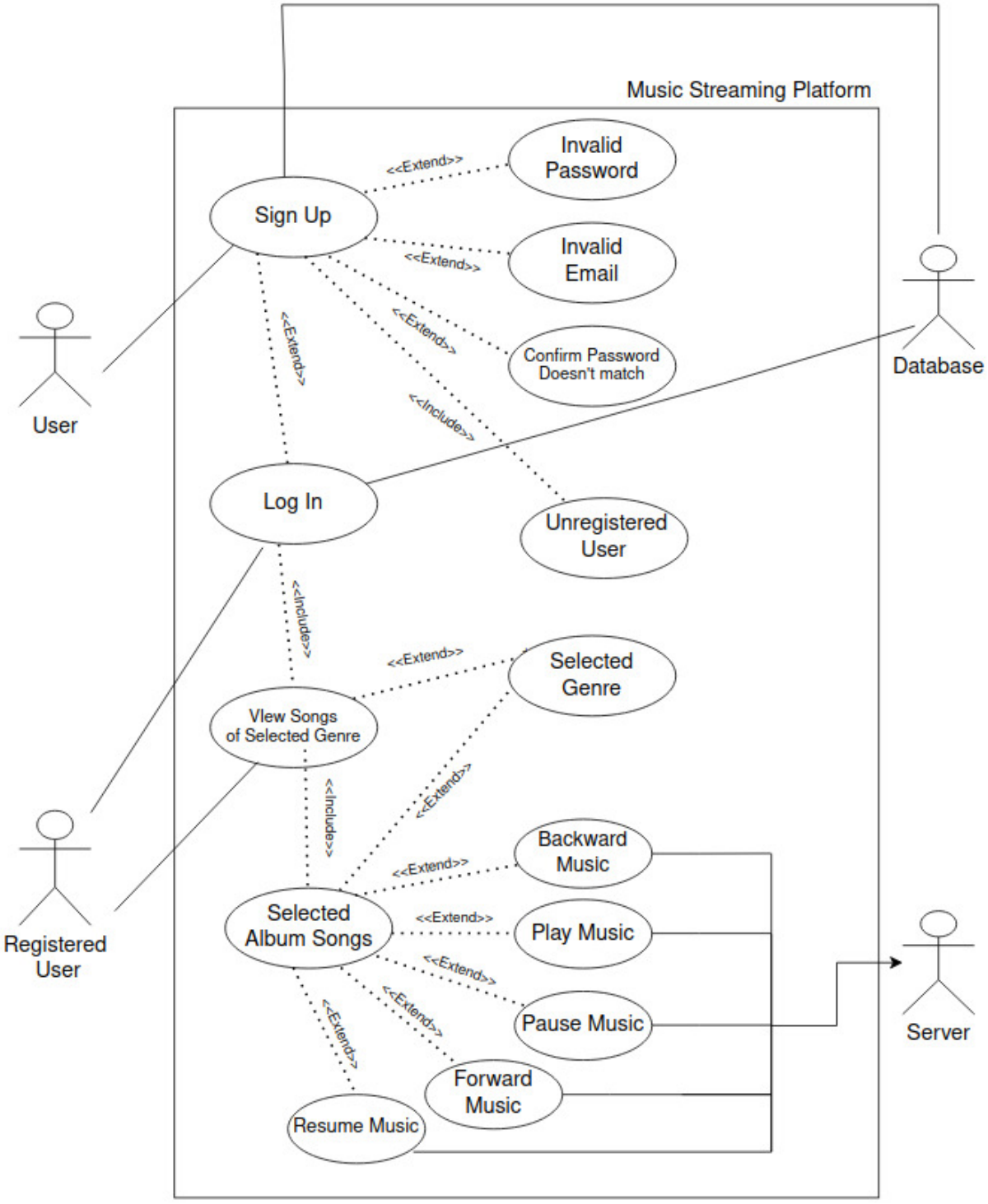
Chapter 3

Design

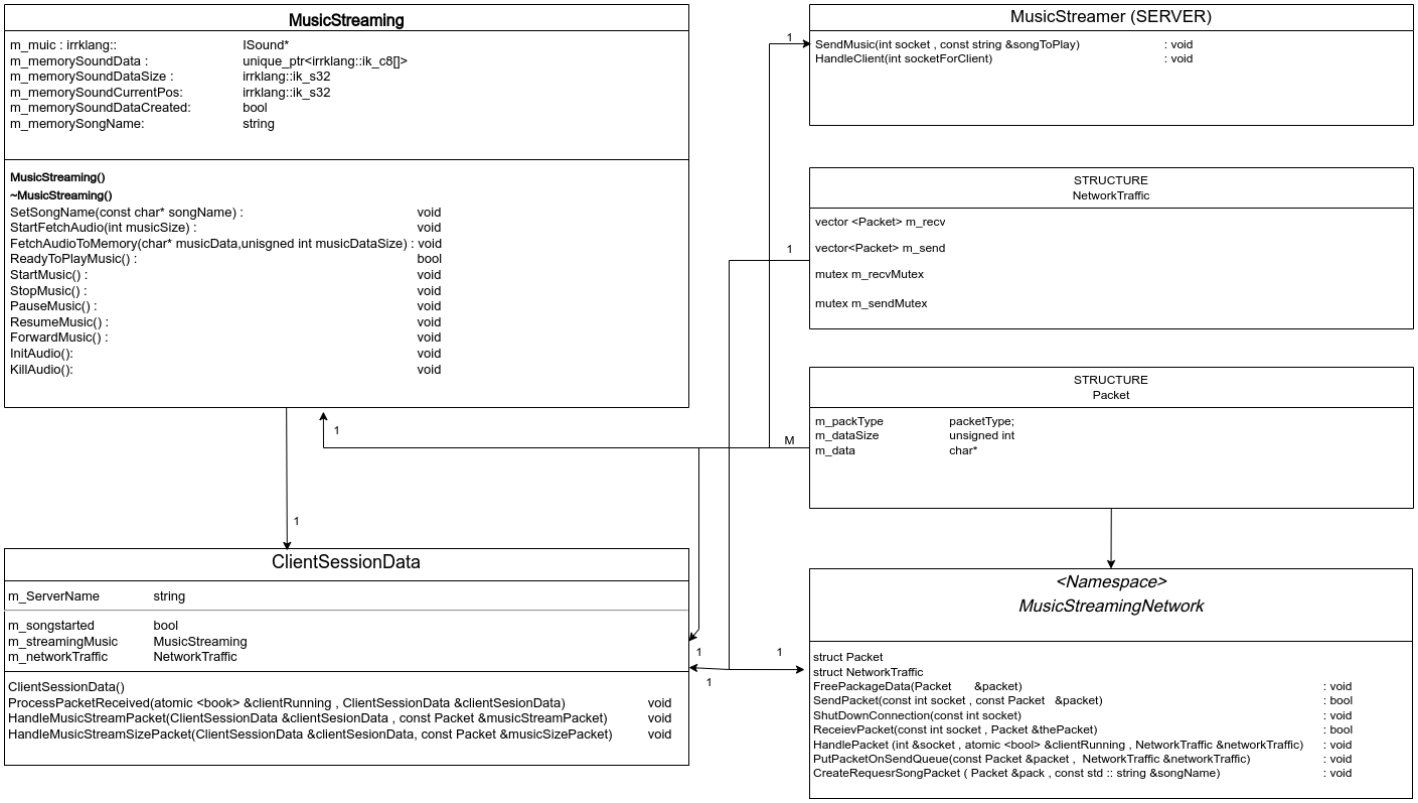
3.1 TCP CONNECTION



3.2 USE CASE DIAGRAM



3.3 CLASS DIAGRAM



Chapter 4

Requirements

4.1 System Requirements

4.1.1 Software Requirements

- Operating System: Ubuntu
- Development Environment: Visual Studio Code (for Ubuntu
- Graphics Library: SFML (Simple and Fast Multimedia Library) , Irrklang
- Database Connectivity: MySQL Connector/C++
- Database Server: MySQL

4.1.2 Hardware Requirements

- Minimum Requirements for Music Streaming Platform:
  - Processor: Intel Core i5 or equivalent
  - Memory: 8 GB RAM
  - Storage: Sufficient disk space for storing music files and application data
  - Network: Stable internet connection for streaming music to clients
  - Sound Card: Compatible sound card for audio playback

Chapter 5

Implementation

Server

`Headerfile : #include <sys/socket.h>`

1. Server Initialization and Configuration

- **Socket Creation**: The server creates a socket using the `socket()` function to establish communication with clients.

```
int serverSocket = socket(AF_INET, SOCK_STREAM, 0);
```

  where,

  - **sockfd**: Socket descriptor, an integer like a file handle.
  - **domain**: Specifies the communication domain, such as `AF_INET` for IPv4 or `AF_INET6` for IPv6.
  - **type**: Defines the communication type, such as `SOCK_STREAM` for TCP (reliable, connection-oriented) or `SOCK_DGRAM` for UDP (unreliable, connectionless).
  - **protocol**: Protocol value for IP, typically set to 0.

- **Bind**: The server binds the socket to a specific IP address and port using the `bind()` function to listen for incoming connections.

```
sockaddr_in serverAddress;
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = INADDR_ANY;
serverAddress.sin_port = htons(PORT);
```

```
bind(serverSocket,reinterpret_cast<sockaddr*>(serverAddress),sizeof(serverAddress))
```

  where,

- **sockfd**: This is the file descriptor of the socket to be bound. It is the socket descriptor returned by the `socket` function.
- **addr**: This is a pointer to a `struct sockaddr` that contains the address and port information to which the socket will be bound. The actual structure passed depends on the address family used (IPv4 or IPv6).
- **addrlen**: This is the size of the address structure pointed to by `addr`. It's important to provide the correct size of the address structure to avoid buffer overflow or truncation.

- **Listen**: The `listen()` function puts the server socket in a passive mode, where it waits for the client to approach the server to make a connection.

  int listen(serverSocket, int backlog)

  where ,
  - **sockfd**: This is the file descriptor of the socket that will be listening for incoming connections. It is the socket descriptor returned by the `socket` function.
  - **backlog**: This parameter defines the maximum length to which the queue of pending connections for `sockfd` may grow. It specifies the maximum number of connection requests that can be queued while the server is busy handling other connections. If the queue is full and a new connection request arrives, the client may receive an error with an indication of `ECONNREFUSED`.

- **Accept**: The `accept()` function is used to accept incoming client connections and establish a new socket connection for each client.

  socketForClient = accept(serverSocket, (struct sockaddr*)NULL, NULL)

  where ,
  - **sockfd**: This is the file descriptor of the socket that is listening for incoming connections. It is the same socket descriptor passed to the `listen()` function.
  - **addr**: This is a pointer to a `struct sockaddr` that will store the address information of the client that is connecting to the server. It is an output parameter and can be set to `NULL` if the address information is not needed.
  - **addrlen**: This is a pointer to a `socklen_t` variable that specifies the size of the `addr` structure. It is an input-output parameter and should initially contain the size of the `addr` structure. Upon successful completion of the function, it will be updated with the actual size of the address structure stored in `addr`.
  - **return** : Creates a new socket with the same socket type protocol and address family as the specified socket, and allocates a new socket descriptor for that socket.

2. Multi-threaded Client-Server Architecture

The server typically employs a multi-threaded architecture where each client connection is handled by a separate thread or process. This allows the server to handle multiple client

connections concurrently without blocking.

```
while (true) {
    SERVER : Waiting for clients to join on port (port number)...
    int socketForClient = ServerWaitForClientToConnect(serverSocket);

    cout << "Client " << countCLient << " joined at socket: " << socketForClient ;
    thread clientThread(HandleClient, socketForClient);
    clientThread.detach();
}
```

3. Inter client server communication using packets

- The structure of Packet contains:
  - `packetType` : { PACKET_REQUEST_SONG , PACKET_MUSIC_STREAM_SIZE , PACKET_MUSIC_STREAM , PACKET_END_CONNECTION }
  - `dataSize` (size of the data)
  - `data` (actual data)
- The Server accepts the packet from client using Packet clientPacket;

```
bool ReceivePacket(const int socket, Packet &thePacket)
```

- The server checks the packetType from received packet . The server receives and processes packets sent by the client, and it sends response packets back to the client as necessary.

4. Sending Music to Client

- One of the main functionalities of the server is to stream music to connected clients upon request. When a client requests a specific song, the server retrieves the corresponding music file, breaks it into packets, and sends the packets to the client for streaming.

```
SendMusic(socketForClient, songRequested);
```

5. Ending Connection on Request

- The server handles client requests to end the connection gracefully. When a client sends a request to terminate the connection (e.g., logout or disconnect), the server closes the corresponding socket connection and cleans up associated resources.

Client

1. Client Initialization and Configuration

- **Socket Creation**: The client creates a socket using the `socket()` function to establish communication with the server.

```
{int sockfd = socket(AF_INET, SOCK_STREAM, 0)}
```

- **Connect to Server**: The client connects to the server using the `connect()` function, specifying the server's IP address and port number.

```
struct sockaddr_in serv_addr;
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(port);

if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    std::cerr << "Invalid address/ Address not supported" << std::endl;
    return -1;
}

if (connect(sock, reinterpret_cast<struct sockaddr*>(&serv_addr), sizeof(serv_a
    std::cerr << "Connection Failed" << std::endl;
    return -1;
}
```

  where,

  - **sockfd**: This is the socket file descriptor, which is obtained from the `socket` function when the socket is created. It identifies the socket to be connected.
  - **addr**: This is a pointer to a `struct sockaddr` that contains the address information (such as IP address and port number) of the remote host to which the connection will be established. The actual structure passed depends on the address family used (IPv4 or IPv6).
  - **addrlen**: This is the size of the address structure pointed to by `addr`. It's important to provide the correct size of the address structure to avoid buffer overflow or truncation. For example, for IPv4 addresses, `addrlen` would typically be `sizeof(struct sockaddr_in)`.

2. User Authentication and User Interface Window

- The server authenticates existing users by requesting their credentials, such as email ID and password, from the client. Subsequently, it validates these credentials to verify the user's identity.
- For new users, the server retrieves their data and conducts basic validation checks before storing it in the database.
- The user interface window enables clients to browse and select songs from a variety of genres.

3. Creating and Sending Requested Song Packet

- When a user selects a song from the interface, the client creates a packet and the secnd the packet to the server containing information about the selected song (song name and

genre).

```
void CreateRequestSongPacket(Packet &pack, const std::string &songName);
void PutPacketOnSendQueue(const Packet &packet, NetworkTraffic &networkTraffic)
```

5. Receiving Music Packets via Threads

- The client receives music packets from the server via separate threads to ensure concurrent handling of music streaming and user interactions.

```
thread threadHandlePackets(MusicStreamingNetwork::HandlePackets, std::ref(socketFor
thread threadProcessPackets(&ClientSessionData::ProcessPacketsReceived, &clientSess
```

6. Playback Controls

- The client provides playback controls such as play, pause, resume, forward, backward, and stop options to allow users to control music playback.

```
void StartMusic();
void StopMusic();
void PauseMusic();
void ResumeMusic();
void forwardSong();
void BackwardSong();
```

7. Quit Connection

- When the user decides to quit the application or disconnect from the server, the client sends a request to the server to terminate the connection gracefully.

```
void KillAudio();
```

Network

1. Create Request Song Packet

- This functionality involves creating a packet containing information about the requested song. The packet typically includes details such as the song name and song type.

```
void CreateRequestSongPacket(Packet &pack, const std::string &songName)
{
    pack.m_packetType = packetType::PACKET_REQUEST_SONG;
    pack.m_dataSize = songName.size() + 1;
    pack.m_data = new char[MusicStreamingNetwork::s_dataSendSize]{};
```

```
    memcpy(pack.m_data, songName.c_str(), songName.size() + 1);
}
```

2. Put Packet on Send Queue

- A queue data structure is used to manage outgoing packets waiting to be sent to the server. When a request song packet or any other type of packet is created, it is placed in this send queue.

```
void PutPacketOnSendQueue(const Packet &packet, NetworkTraffic &networkTraffic)
{
    std::lock_guard<std::mutex> guard(networkTraffic.m_sendMutex);
    networkTraffic.m_send.push_back(packet);
}
```

3. Send and Receive Packet

- This functionality handles the transmission of packets between the client and the server. The client sends packets to the server, and the server sends response packets back to the client.

```
bool ReceivePacket(const int socket, Packet &packet);
bool SendPacket(const int socket, const Packet &packet);
```

4. Using Send and Receive via Threads

- To ensure concurrent handling of packet transmission and other client activities, sending and receiving packets are often performed in separate threads. This allows the client to continue its operation without being blocked by network I/O.

```
thread threadHandlePackets(MusicStreamingNetwork::HandlePackets, std::ref(socketFor
```

5. Process Packet

- Upon receiving a packet from the server, the client processes it based on its type. For example, if the packet contains music data, the client may play the music. If it's a response to a request song packet, the client may update its user interface or take other appropriate actions.

```
thread threadProcessPackets(&ClientSessionData::ProcessPacketsReceived, &clientSess
```

6. Free Packet

- After processing a packet or when it's no longer needed, the client frees the memory occupied by the packet to avoid memory leaks.

```
        void FreePacketData(Packet &packet)
    {
        if (packet.m_dataSize > 0)
        {
            delete[] packet.m_data;
        }
    }
```

Audio

1. Creating Engine

- This functionality involves creating an audio engine that is responsible for playing music and managing audio-related operations.

```
    void StreamingMusic::InitAudio()
  {
   // start the sound engine with default parameters
   s_engine = createIrrKlangDevice();

   if (!s_engine)
     {
      cout << "Could not init audio" << endl;
   }
  }
```

2. StartFetchAudio

- When a request to stream music is received, this functionality initiates the process of fetching audio data. It may involve setting up buffers or allocating memory for audio streaming.

```
        void StreamingMusic::StartFetchAudio(int musicSize)
{
    m_memorySoundDataCreated = false;
    m_memorySoundDataSize = musicSize;
    m_memorySoundData = std::make_unique<irrklang::ik_c8[]>(m_memorySoundDataSize);
    m_memorySoundCurrentPos = 0;
    cout<<"Fetched Audio"<<endl;
}
```

3. ReadyToPlayMusic

- This function checks if the audio data is ready to be played. It may verify whether a sufficient amount of audio data has been fetched to start playback.

```
        bool StreamingMusic::ReadyToPlayMusic() const
{
    return m_memorySoundCurrentPos > 500; // 500kb
}
```

4. FetchAudioToMemory

- This functionality fetches audio data from the server and stores it in memory. It may involve reading audio packets from the network and buffering them for playback.

```
        void StreamingMusic::FetchAudioToMemory(char* musicData, unsigned int music
{
    memcpy(m_memorySoundData.get() + m_memorySoundCurrentPos, musicData, musicDataS
    m_memorySoundCurrentPos += musicDataSize;

    if (!m_memorySoundDataCreated)
    {
        cout << "Adding sound source " << m_memorySongName.c_str() << endl;
        ISoundSource* source = s_engine->addSoundSourceFromMemory(
            m_memorySoundData.get(),
            m_memorySoundDataSize,
            m_memorySongName.c_str(),
            false
        )
}
m_memorySoundDataCreated = true;
}
```

5. AddSoundSourceFromMemory

- Once audio data is fetched to memory, this function creates a sound source from the memory buffer. It allows the audio engine to play the music directly from the memory buffer.

```
ISoundSource* source = s_engine->addSoundSourceFromMemory(
            m_memorySoundData.get(),
            m_memorySoundDataSize,
            m_memorySongName.c_str(),
            false
        );
```

6. StartMusic

- After setting up the audio source, this function starts playing the music. It initiates playback of the audio data fetched from memory.

```
        void StreamingMusic::StartMusic()
{
    cout << "Playing " << m_memorySongName.c_str() << endl;
    m_music = s_engine->play2D(m_memorySongName.c_str(), true, false, false, ESM_ST
    m_music->setVolume(0.5f); // Personal preference here, I turn the volume down t
}
```

7. StopMusic

- This function stops the playback of music. It may be called when the user pauses the music or when the music streaming session ends.

```
       void StreamingMusic::StopMusic()
{
    if(m_music)
    {
        cout << "Stoping music" << endl;
        s_engine->removeSoundSource(m_memorySongName.c_str());
        m_music->drop();
    }

}
```

8. PauseMusic

- This function pauses the playback of music. It allows the user to temporarily halt the music playback without stopping it entirely.

```
       void StreamingMusic::PauseMusic()
{
    if (m_music)
    {
        cout << "Pausing music" << endl;
        m_music->setIsPaused(true);
    }
}
```

9. ResumeMusic

- This function resumes the playback of music after it has been paused. It allows the user to continue listening to the music from where it was paused.

```
        void StreamingMusic::ResumeMusic()
{
```

```
    if (m_music)
    {
        cout << "Resuming music" << endl;
        m_music->setIsPaused(false);
    }
}
```

10. ForwardMusic

- This function forwards the playback of music to the next track or a specified time point. It allows the user to skip forward in the music playback.

```
void StreamingMusic::ForwardMusic()
{
    if (m_music)
    {
        // Get the current playback position
        int currentPosition = m_music->getPlayPosition();

        // Forward by 10 seconds (10000 milliseconds)
        int newPosition = currentPosition + 10000;

        // Set the new playback position
        m_music->setPlayPosition(newPosition);
    }
}
```

11. BackwardMusic

- This function rewinds the playback of music to the previous track or a specified time point. It allows the user to skip backward in the music playback.

```
       void StreamingMusic::BackwardMusic()
{
    if (m_music)
    {
        // Get the current playback position
        int currentPosition = m_music->getPlayPosition();

        // Forward by 10 seconds (10000 milliseconds)
        int newPosition = currentPosition - 10000;

        // Set the new playback position
        m_music->setPlayPosition(newPosition);
    }
}
```

12. Play2D

- The `play2D` function of the irrKlang library is used to play a sound or music file in 2D (i.e., with no spatial positioning, like stereo sound).

  m_music = s_engine → play2D(
  m_memorySongName.c_str(), true, false, false, ESM_STREAMING, true)

  where ,

  - **soundFileName**: The path or name of the sound or music file to be played.
  - **playLooped**: Optional parameter to specify whether the sound should be played in a loop. Default is `false`.
  - **startPaused**: Optional parameter to specify whether the sound should start in a paused state. Default is `false`.
  - **track**: Optional parameter to specify whether the sound should be tracked. Default is `true`.
  - **mode**: Optional parameter to specify the stream mode for the sound. Default is `ESM_AUTO_DETECT`.
  - **enableSoundEffects**: Optional parameter to specify whether sound effects should be enabled. Default is `false`.

Chapter 6

Test Cases

1. **Successful Connection of Multiple Clients to One Server:**
   Verify that multiple clients can connect to a single server without encountering connection issues.
   For verify click : **(1)**

2. **New Client Registration and Data Entry Creation:**
   Test the registration process for new clients followed by the successful creation of new data entries in the database.
   For verify click : **(1)** **(2)**

3. **Successful Login for Existing Clients:**
   Ensure that existing clients can log in to the system without any authentication issues.
   For verify click : **(1)**

4. **Basic Credential Validation at Login and Signup:**
   Perform basic checks on login and signup pages to validate user credentials and ensure security measures are implemented.
   Passed

5. **Genre-Based Song Filtering:**
   Validate the functionality to filter songs based on their genre to ensure accurate and efficient filtering.
   Passed

6. **Client Termination by Pressing Quit Button:**
   Verify that clients can successfully close the application by pressing the "Quit" button without any unexpected errors.
   Passed

7. **Sequential Music Streaming by Multiple Clients:**
   Test the ability of multiple clients to stream music sequentially without interruptions. Check for edge cases where some clients may encounter issues while others do not.
   Partially Passed (Will not work until server send a full music file)

8. **Server unexpectedly Terminate :**
   Test by connecting client to server and then unexpectedly terminate the server
   Partially Passed (Will not work for if we streamed another song)

9. **Kill Server Port**
   Test by connecting client to server and then killing the server port
   Partially Passed (Will not work for if we streamed another song)

Chapter 7

Valgrind Report

1. **Server Valgrind Report**

   Valgrind logs file : **Click Here**

2. **Client Valgrind Report**

   Valgrind logs file : **Click Here**