Audio Streaming Service

Emiliano Xhukellari

0. Table of Contents

Contents

1. Introduction

This project aims to build an audio streaming service that is convenient to use and provides high-fidelity (HiFi) audio for users over the network. This audio streaming service consists of two parts – Client Application and Server Application. These two applications communicate with each other using the TCP protocol.

The technologies I am using for this project are all available on any Desktop or Laptop computer running Windows OS. I have chosen to use C# because it is a fast and convenient language that supports GUI creation. I have chosen WPF Application because it supports a broad set of application development features – application model, resources, controls, graphics, layout, data binding, document, and security. WPF also uses the Extensible Application Markup Language (XAML) that provides declarative modeling for the application programming (“Windows Presentation Foundation - WPF .NET Framework”).

Apart from the default C# technologies, I am using the WaveOut API which comes with any Windows computer (XP or later). This is a low-level library that will enable my Client Application to play wave files in any existing output devices in the computer. I have built my own wrapper for WaveOut API in C# that can make calls to native windows functions. I tested multiple audio libraries that are written in C# that can play various forms of audio files, but none of them provided the low-level functionalities that are necessary to make the audio streaming service efficient and responsive. Most audio libraries expect the programmer to provide a full stream of audio data. As we will see later in this paper, our client and server applications are designed to play audio responsively – not only will the users be able to play HiFi music, but they can also do so without the waiting time involved in sending the audio data to the client.

Furthermore, in the development of this project, I have extensively used the TCP protocol implemented in .NET. This protocol makes communication between the Server and multiple clients possible.

I use applications like Spotify, Apple Music, and Tidal every day because I love listening to music, especially high-quality music where you can hear the details in the song – vocals, instruments, etc. I strongly believe that music is one of the most expressive forms of art that can make others feel emotions they would never feel otherwise. Some people listen to music to help them study, sleep, run, etc. Other people, on the other hand, take music to the next level by using special headsets and paying premium prices to get the highest quality music. There is indeed a difference between the quality the user will experience when listening to songs provided by Spotify and Tidal - the latter provides significantly higher quality music. The audio streaming service that I am working on also provides HiFi music, and it does so with high responsiveness.

I enjoy building desktop applications that involve multi-threading and multi-processing. An audio streaming service is a great candidate to test my skills in the field of multiprogramming, working with memory, low-level calculations with bytes, etc. To some extent, the client and server applications might look over-engineered, but I plan to keep improving these applications in the future; thus, the applications need to be as efficient and effective as possible. In the following pages, I will describe in detail how I managed to build a working audio streaming service that can be used by anyone that owns a Windows PC running Windows XP or later.

2. Specifications of the Software Requirements and Analysis

First, I will start with the functional requirements for the client application. The targeted users are those who like to listen to music and have the bare minimum of knowledge on how to use computer software such as Spotify, Tidal, Apple Music, etc. They need not have any special skills to use this software. Thus, it must be as simple to use as possible and, at the same time, provide all the functionalities that a music player must have. Also, the client application should be able to play podcasts if the user chooses to stream podcasts.

2.1. Client Application Functional Requirements

1. The client application must be able to play wave files using a windows PC running Windows XP or later (I will explain later what a wave file is).
2. The client application must be able to establish TCP connections with the server.
3. The client application must be able to reconnect with the server in case of disconnection.
4. The client application must support all basic commands of a music player.
    a. It must support the play song command.
    b. It must support the pause song command.
    c. It must support the next song command.
    d. It must support the previous song command.
    e. It must support the volume-up command.
    f. It must support the volume-down command.
    g. It must support shuffling songs in the queue command where songs are randomly swapped and not repeated more than they were previously repeated before shuffling.
    h. It must support unshuffled all songs in the queue, where the effects of shuffle are reverted.
    i. It must support the search for a song command, where the user starts typing text and songs are displayed and ready to be played, added to the queue, and added to playlists.
    j. It must support playlist creation, deletion, and renaming.

When it comes to the server, the user I am targeting is someone who has some knowledge of types of files such as WAVE files and PNG files. Apart from that, anyone can use this server application. They should not need special skills, such as knowing how to work with databases, SQL, etc. The server application should also be relatively easy to use. The functional requirements for the server are the following.

2.2. Server Application Functional Requirements

1. The server application must be able to support multiple clients simultaneously.
2. The server application must be able to send song information to the client. This information includes the song or podcast name, the artist name, the duration of the song, and an image representing the song or podcast.
3. The server application must be able to stream songs to the client.
4. The server application must be able to add songs to its database.
5. The server application must be able to establish TCP connections with the clients.
6. The server must be able to reconnect with clients in case they disconnect.

Next, we continue with the non-functional requirements for the client application and the server application. We will start with the client first.

2.3. Client Application Non-Functional Requirements

1. The client application must be responsive while the application is running. The UI must not be blocked regardless of the operations happening in the background.
2. The client application needs to have a smooth playback. We are working with wave files, which are not compressed and are relatively large. This could affect the playback of the sound. Thus, the client application must be able to provide smooth playback even when the wave files are very large (up to 2GB). If the client moves the progress of the audio, it should start playing within 500ms.
3. The time it takes from the user clicking play, next, previous, or any other action that plays audio to music being played from the speakers must be less than 500ms.

As you can see, the client application’s non-functional requirements focus on making the experience of the user as good as possible. When one uses an audio streaming application, they prefer to have a responsive GUI, responsive sound, etc. – no one likes to wait for their songs to start playing. Next, we will continue with the server.

2.4. Server Application Non-Functional Requirements

1. When a song request happens, the server must be able to start sending data immediately. All operations that are not dynamic must be done beforehand. This will increase responsiveness on the client side and lower resource use on the server.
2. The server should use memory resources efficiently. If a client is disconnected, release any memory and processing power so that the server can serve as many clients as possible.
3. Adding songs to the server application storage should not require any special skills. The process should be as easy as inputting data files regarding song name, artist name, choosing files with native Windows dialogs, etc.

Now that we have finished with specifying the functional and non-functional requirements for the client and server application, we will continue with the analysis part. We will start with the analysis of functional and non-functional requirements of the client and continue with the functional and non-functional requirements of the server.

2.5. Analysis of Functional and Non-Functional Requirements

1) The client application must be able to play wave files using a windows PC running Windows XP or later. (Functional)

2) The client application needs to have a smooth playback. We are working with wave files, which are not compressed and are relatively large files. This could affect the playback of the sound. Thus, the client application must be able to provide smooth playback even when the wave files are very large (up to 2GB). If the client moves the progress of the audio, it should start playing within 500ms. (Non-Functional)

Wave files are popular, and Windows supports playing wave files natively ("Wave and DirectSound Components - Windows Drivers"). However, it would not be smart to isolate this functional requirement without considering one important non-functional requirement – the client needs to have smooth playback. Since we will be using C# for the development of the audio streaming service, we need to do excessive research on what libraries can help us achieve both functionalities. The functional requirement of our client application being able to play wave files is very straightforward – we need to find a way to play HiFi wave files using a PC that runs Windows XP or later. However, the smooth playback non-functional requirement makes the situation more complex. To provide smooth playback for the user regardless of the size of the wave file, we should consider the following:

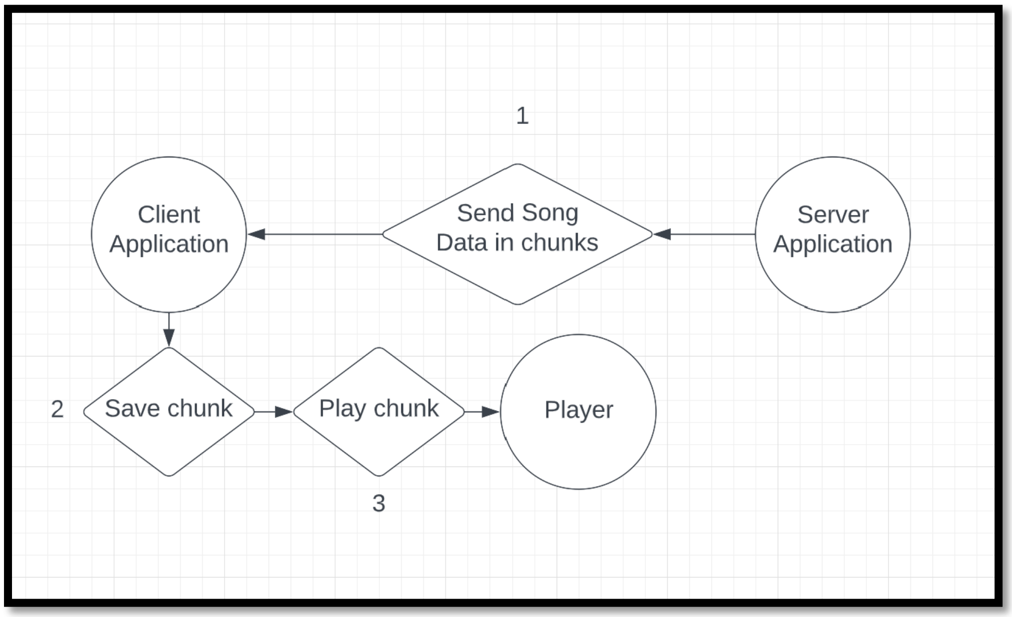
1) Most users will have a relatively slow internet connection. The average home internet speed in Europe in 2021 was reported to be 38 Mbps (Fair Internet Report). A wave file for a song 3-4 minutes is around 40-50 MB or 320 Mb. This means that to receive the entire song it will take:

320 Mb / 38Mbps = 8.4 seconds

8.4 seconds of wait-time is unacceptable when it comes to smooth playback. The user should not have to wait 8.4 seconds after clicking the play button for the sound to start playing. Therefore, it is essential for our client application to not only play wave files but also start playing them immediately after the user clicks play. This means that the client must have an internal player that plays sound in chunks and does not require the full wave file.

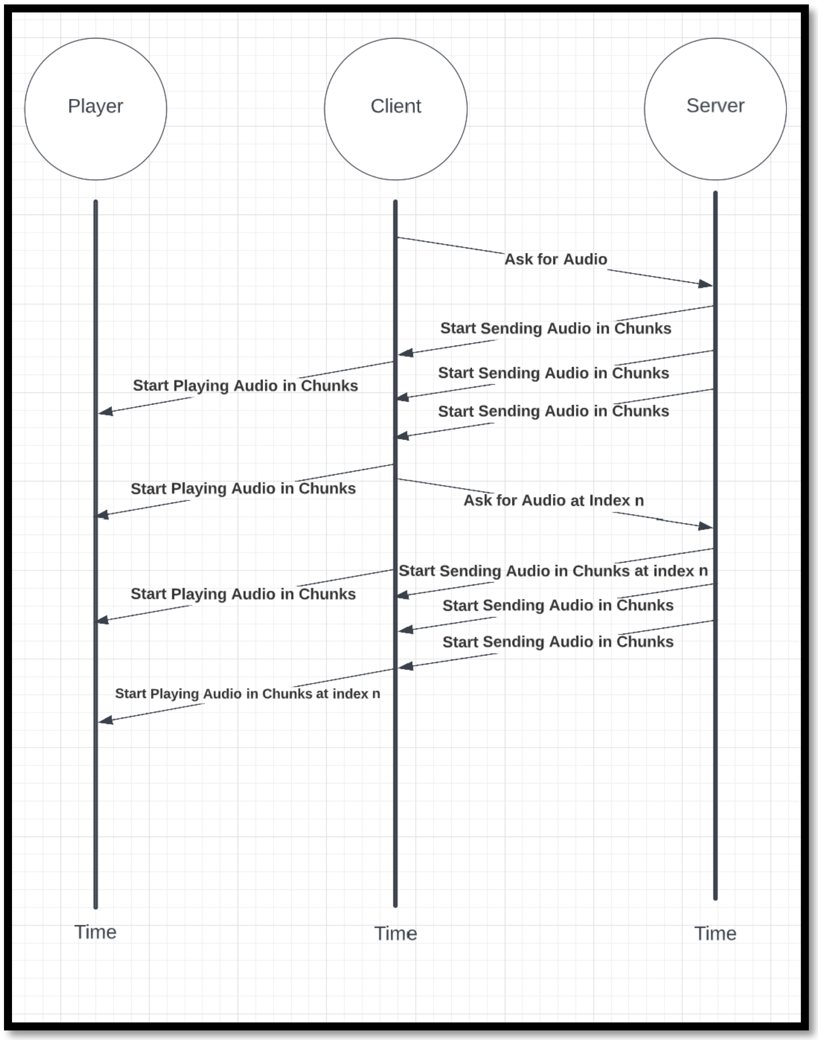
2) The users might decide to play podcasts. Usually, podcasts are quite long. They might be a couple of hours long. This means that the wave files can reach a couple of gigabytes in size.

Therefore, the client application must have low-level control over how wave files are played by the computer. The following diagram illustrates what our client application must be able to do:



Client Application Playing Sound Data Supplied via the Network

The server will start sending wave files in chunks using the TCP protocol, and the client application will start playing the chunks immediately without waiting for the entire data to be received. Furthermore, in the non-functionality of the smooth playback, it is mentioned that if the user changes the progress of the audio, the player should start playing sound immediately within 500ms. This requirement means that the client application should react as if it has the entire audio received, even if it does not. Thus, we need to find a way for the client to immediately ask the server for that specific position of the audio file. For this to be done, the client needs to inform the server that it no longer needs data at the current index, but instead it needs data at index n. Afterward, the client should wait for this data to be sent by the server. The server application will understand the request and it will start sending data at that index. In this paper, I will refer to this concept as Dynamic Streaming. It is dynamic because the client-server interaction is not predetermined. The client can tell the server at any time to send a specific amount of data at a specific index. All these interactions are described in the following diagram:


Player
Client
Server
Ask for Audio
Start Sending Audio in Chunks
Start Playing Audio in Chunks
Start Sending Audio in Chunks
Start Sending Audio in Chunks
Start Playing Audio in Chunks
Ask for Audio at Index n
Start Sending Audio in Chunks at index n
Start Playing Audio in Chunks
Start Sending Audio in Chunks
Start Sending Audio in Chunks
Start Playing Audio in Chunks at index n
Time
Time
Time

Client-Server Interaction

In the diagram we can see the following: First, the lines representing data being sent from the server repeat more than the lines being played by the player. This is necessary because the amount of data we receive should be higher or equal to the amount of data that is being played. Only this way can we have smooth playback – the player needs data to be fed consistently. Second, if the client asks the server to send data at index n, the server will start sending data from that index. At the same time, the client application instructs the player or another component that supplies data to start reading data from that index. This is how we reach smooth playback regardless of what commands the user inputs. For instance, if the user clicks the play button, the song will start playing immediately since the client application does not need to wait for the entire song to be received – it only needs enough data to fill the buffers in the player. If the user decides to skip to the middle of the song or podcast, the client application does not need the specific index to be received – it will instead ask the server to send data to that index. After some research, I have decided to use wave/out for the player provided by Microsoft, which can be used on any Windows PC running Windows XP or later without any additional software. The wave/ out class consists of audio devices for low-level wave audio output (“Wave/Out - Win32 Apps”).

Next, we will continue with the second and third functional requirements of the client and the fifth and sixth functional requirements of the server.

3) The client application must be able to establish a TCP connection with the server. (Functional Client)
4) The client application must be able to reconnect with the server in case of disconnection (Functional Client).
5) The server application must be able to establish TCP connections with the clients. (Functional Server)
6) The server must be able to reconnect with clients in case they disconnect. (Functional Server)

These requirements are very straightforward. Our client needs to use the TCP protocol, which C# supports by default (“System.Net.Sockets Namespace”). The safe applies to the server. This requirement is here because the TCP protocol has numerous features which will make our audio streaming service possible. First, the TCP protocol is connection-based – the client and server need to establish a connection before any data is sent. Second, the TCP protocol is reliable – all data that is sent will be received. Third, the TCP protocol provides flow control – the server will not flood the client with too much data (Eddy). However, we need to make some clarifications on this. We know that wave files are relatively large, and the socket that will be transporting this data will be utilized for a relatively long time. We also know that the client and the server need to exchange information regularly – for instance, when the client needs to tell the server where to start sending data. Therefore, it is best two equip our client and server with two dedicated sockets – one for streaming audio data, and one for exchanging communication messages. Furthermore, we need to design our sockets so that they can reconnect in case of disconnection. Since we will have a dual-socket design, the connection and reconnection of the sockets need to be parallel – if one of them disconnects, we should reassign two new sockets.

Next, we have the fourth functional requirement of the client:

7) The client application must support all basic commands of a music player.

If we pay close attention to the basic commands that are required here, we can make some separations. Some of these functions are supported natively by wave /out, although we need to have a C# wrapper for it.

Play, pause, stop, and volume control is supported by wave/ out by using native windows commands. When the user clicks play, we need to communicate to wave/ out that it needs to start playing audio; when the user clicks pause, we need to communicate to wave/ out that it needs to pause the audio; when the user changes the volume, we also need to tell the wave/ out the changes.

However, the other commands are higher-level commands, and we need to have other components that will deal with them. When the user clicks next song, our client application needs to stop playing the current audio and start playing the other one. The same applies to previous song.

Shuffle is a command that can be interpreted in different ways. In our case, we will make sure that songs are ordered randomly but they do not appear more than they already did before shuffling. For instance, if we have the songs A, B, C, and D in the queue and we decide to shuffle, we could get B, A, D, and C. However, we cannot get A, A, D, B – here A has repeated more than once.

The unshuffle command, on the other hand, should undo what the shuffle did. B, A, D, and C, should go back to the previous state A, B, C, D.

The repeat command refers to what happens when the end of the song is reached. There are usually three states of repeat in most media players – Repeat One Song, Repeat All Songs, and Do Dot Repeat. If we have Repeat One Song, then the player should keep playing the same song repeatedly until the user clicks the next song or previous song. If we have Repeat All Songs, the player should start playing the songs from the beginning when the end of the queue is reached. If we have Do Not Repeat, the player should keep playing the last song in the queue.

Searching for songs means that the user will be provided with a text box where they can search for a song, podcast, or artist. The moment the user enters some data, the client application should ask the server for the song, podcast, or artist, and it should receive some results.

When it comes to playlists, the client application should be able to do the following:

1) Create a new playlist
2) Delete a playlist
3) Rename a playlist
4) Add songs or podcasts to the playlist

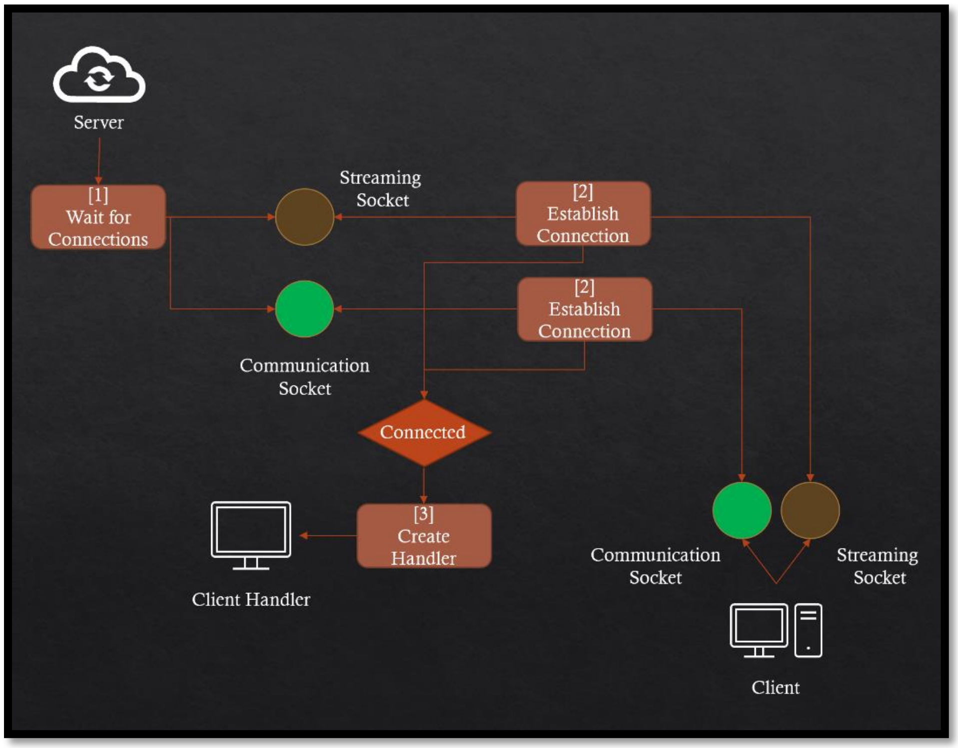
The user should be displayed an option to create a new playlist, provide a name for the playlist, and create the playlist. Furthermore, for each playlist, the user should be provided with the option to delete this playlist. When deleting a playlist, the client application needs to delete the playlist and all the songs in that playlist. Also, the client must provide the user with the option to rename a playlist. The user clicks on a button to remove the playlist, the application provides the user with a text box to enter the new name, and the client application renames the playlist. Finally, the user should be able to add songs that they search for in the playlist. With the click of the button on a specific song, that song should be stored in the client storage for further searches from the playlist section.

We will now continue with the analysis of the remaining functional requirements of the server application.

The first functional requirement of the server that we have not analyzed yet is the following:

8) The server application must be able to support multiple clients simultaneously. (Functional Server)

It is crucial for the server to support multiple clients simultaneously because if the audio streaming service supports only one client, the entire service would be pointless. Since we are using the TCP protocol, this should be possible. We need to design the server in a way that it waits for client connection requests, accepts the requests, and then assigns a handler for the specific clients. Each handle can be run in a separate parallel unit – in C# we can use .Net Threads. They are efficient and can run in parallel. The constraints will be how much processing power the server computer has. The following diagram describes how this requirement will be achieved:



Assigning Client Handlers

In the diagram, we can see that the server waits for connections in two sockets, accepts the connections, and assigns a client handler. This client handler is run in a separate thread, and the server keeps listening for connections in the main thread. Thus, we have met this functional requirement.

The second functional requirement for the server is:

9) The server application must be able to send song information to the client. This information includes the song or podcast name, the artist name, the duration of the song, and an image representing the song or podcast.

When the client searches for a song or podcast, we should not send the wave data yet. Instead, we will send the data that is described in this functional requirement. These properties identify the audio file, and the server must be able to send this information to the client. This can be achieved using the TCP protocol running in a separate communication loop that does not interfere with the streaming TCP socket. All these properties can be converted into bytes, and then sent to the client. The process is the following:

0. Create a song object after retrieving the song name, the artist name, the duration, and an image from the database.
1. Serialize this object using a function from the song object that converts all these attributes into bytes.
2. Slice the bytes into chunks with appropriate sizes for the TCP sockets, such as 1024, 2048, etc.
3. Get the number of packets and send the number to the client.
4. Send the packets to the client.

When the client application receives all the packets, it can convert the bytes back into a song object using its constructor.

The third functional requirement for the server is:

10) The server application must be able to stream songs to the client.

We can understand this requirement as the server must be able to send wave files to the client using the TCP protocol. Wave files are already in bytes, so we can simply read the bytes, slice them into chunks, and send them to the client. However, instead of doing that, we should prepare the wave file for Dynamic Streaming. For Dynamic Streaming to work we need to know the index of each packet. Therefore, we can do the following:

1. Perform any processing on the wave file to serialize it and get it prepared to be used by the client.
2. Get the header and save it – this header is essential for playing wave data.
3. After the header, we get access to the data of the wave file. Here we can slice the data into chunks.
4. Before each chunk add the index of the chunk.

Using the TCP protocol, we can send chunks of an appropriate size, such as 4096.

The final functional requirement of the server is:

11) The server application must be able to add songs to its database.

As we previously mentioned, a song has numerous properties – song name, artist name, duration, and image. Also, there is a wave file associated with each song or podcast. To make our database efficient, we can save large files (wave files and image files) in specific folders in the server and add in the database file only the path names. For the server user to add songs to the database, we need to provide the necessary input fields for them to do so. We need to provide a text box for the song name, and artist name, an open file dialog for the wave file, an open file dialog for the image file, and a button to add the song to the database. The input needs to be validated that it is complete and that it is ready to be stored in the database. The process of adding a song to the database will be the following:

1. The server user enters the song name
2. The server user enters the artist name
3. The server user selects the wave file
4. The server user selects the image file
5. The server user clicks add to database button
6. The server checks that the input is correct and complete
7. The server serializes the wave file and the image file
8. The server renames those files into appropriate files that can be registered in the database in the following format – {Song Name} by {Artist Name}.wav/.png
9. The server enters the data into the database table that contains the songs

3. Design of the Software Solution

The main architectural pattern that I have used for this project is the Client-Server architecture pattern. This architectural pattern is the most suitable for distributed applications that will be run on different networks (“Anatomy of the Client/Server Model”).

The client requests specific resources to be fetched by the server. In our case, the client might request a song, images, audio, text, etc.

The client is an application that will be run by the user on their personal computers. This application is designed to perform multiple tasks such as connecting to audio devices in your computer, playing audio, maintaining a queue of songs, etc. In fact, most of the logic of this audio streaming service is programmed in the client application for various reasons – some of these reasons are privacy, efficiency, separation of concerns, etc. However, the client application alone cannot perform its main duty – playing audio. The client needs audio data that can only be accessed from the server application.

The server, on the other hand, has one main job – to provide clients with whatever data they need. Usually, clients will request some songs based on what they search for. The server finds possible songs that match the pattern sent by the client and send those results.

Both, the client application, and the server application are quite complex. They need to provide many functionalities and they need to be robust. Therefore, it is important that both applications are structured well. We need to separate different functionalities for both applications so that they can be easily maintained and easy to work with.

3.1. Server Side

Sever Application's main job is to server the client applications. Its main job is to supply audio data to the clients. The server application has three main components: Controller Class, Client Handler Class, and Database. The Controller class is the second highest level class after the Application Class which is managed by .NET. Aggregation is the main form of relationship between classes in the Server Application. The Application Class owns an instance of the Controller Class; the Controller Class owns many instances of the Client Handler Class.

3.1.1. Server Controller Class

As the name suggests, this class's responsibility is to control the flow of the application. The Controller Class is the heart of the application. A single instance of this object is created by the WPF Application when the Server application starts, and by overriding the OnStartup() method, we can start the controller. To start the Controller we need to call its Run() method. The Controller object is multithreaded, and the Run() method is non-blocking. It will simply start all the threads in the controller and continue running the main thread.

The Controller owns two dedicated threads that are listening for TCP connection requests. These two functions are running independently of one another and thus having low coupling. As previously mentioned, the audio streaming service uses two sockets – communication and streaming socket. When the server accepts a connection for communication, the socket will be stored by the server. The same happens for the streaming connection. If the server finds two sockets of two different types (streaming and communication) that are connected to the same client, the server will then create a client handler object.

3.1.2. ClientHandler Class

Objects of this class are only created by the controller the moment the controller has two sockets – one for streaming and one for communication – and a database connection. A ClientHandler can perform different functionalities, and all these functionalities are separated into different methods and classes. For instance, one of the responsibilities of the ClientHandler is to supply Songs to the client when requested. This functionality is separated from other functionalities such as sending wave song data to the client when requested. Two different loops run in separate threads which also have separate functionalities and use different sockets. First, we have the Communication Loop which exclusively uses the communication socket that was supplied by the controller. This is a function that is run in a separate thread, and its task is to execute communication requests. There are two communication requests that a ClientHandler object can perform:

1) Search request
2) Terminate Song Data Request

A search request has the format “SEARCH@{input}”. A search request will look in the database for two attributes – Song Name Serialized, or Artist Name Serialized. Based on the result, the Server will send the number of songs it found that match the string pattern that was sent with the search request. Thereafter, the ClientHandler will create Song objects that contain the attributes that describe the song such as Song ID, Song Name, Artist Name, Duration, and an Image in binary. These Song objects are serialized into bytes using the method GetSerialized() that the Song Class contains (this is very convenient because we have a quick way of converting Song objects into bytes, and bytes into Songs - TCP sockets can only supply bytes). Then, the bytes object is sliced into chunks and finally sent to the Client. The client will then have to deserialize the object and convert it into a Song object.

A terminate song data receive is used by the client handler to determine whether it should stop sending wave data. As we previously mentioned, the goal of this audio streaming service is to provide HiFi Audio, be responsive, and efficient. If the client application needs to skip a song, or the client application needs some data at another index, the server needs to have a way of switching tasks. This is done by using this type of request. In other words, we are using the communication loop and communication socket to control the streaming loop and the streaming socket.

Second, the ClientHandler has a streaming loop that exclusively uses the streaming socket. This is where most utilization of the ClientHandler happens since the socket is sending big amounts of wave data. The streaming loop is intelligent in its own way. Not only can it tell the client application whether it is sending data or requesting a termination, but it can also recognize the mode of streaming. There are two modes of streaming –

MOD1 which represents standard streaming (starts sending data from beginning to end), and MOD2 which represents specific range streaming (starts sending data from index A to index B, which are sent by the client application). MOD2 is used for Dynamic Streaming. SendSongData() is the method that is used to perform the streaming of data. If either of the sockets disconnects, the ClientHandler object will terminate itself. Also, if one of the sockets disconnects, the other one will disconnect too. This is because the client user closed the client application. Now, the ClientHandler object needs to release the memory by notifying the Controller.

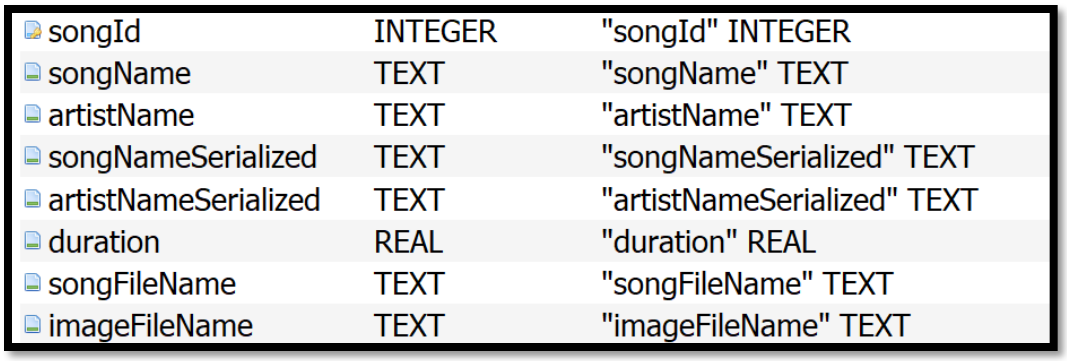
3.1.3 Custom Server Events

I previously mentioned that the ClientHandler needs to release memory and notify the Controller. But the ClientHandler is an object that is owned by the Controller. How does a class that is owned by the Controller notify the Controller? This is done using the Custom Event Classes – ServerEvent and Listener.

When a higher-level class needs to listen for events from a lower-level class, the higher-level class needs to declare a Listener object. In our design, I am using composition so that the Controller can listen for server events. By calling the Listen() method the Controller subscribes to server events of a specific type. To invoke a server event, the caller simply needs to create a new instance of a server event and specify the type. Afterward, the server event will know which specific method to call based on what the subscriber has specified.

Now that we have described the design of the most critical parts of the server application, we can continue to explain how the server is designed so that the user can easily use all its features.

When the server user executes the server application, they are provided with a simple user interface. Here the client user can see which clients are connected. Furthermore, the client user can add songs to the database. There are two text fields that need to be supplied – Song Name and Artist Name. Also, there are two files that need to be supplied – Song File and Artist File. When the server application was started, it already either created a new database file if it was not there, or it opened one that was there.



songId	INTEGER	"songId" INTEGER
songName	TEXT	"songName" TEXT
artistName	TEXT	"artistName" TEXT
songNameSerialized	TEXT	"songNameSerialized" TEXT
artistNameSerialized	TEXT	"artistNameSerialized" TEXT
duration	REAL	"duration" REAL
songFileName	TEXT	"songFileName" TEXT
imageFileName	TEXT	"imageFileName" TEXT

Songs Database Table

The database for the server is quite simple. It contains only one table that stores the necessary information about each song. When the client user adds a new song, songId, which is the primary key, is incremented by 1. In this table, you can see some additional fields that were automatically generated by the server.

songNameSerialized – this field is the result of removing white spaces from songName.

artistNameSerialized – this field is the result of removing white spaces from artistName.

duration – this field is calculated using the information in the audio file.

songFileName – this is the relative path to the newly generated audio file.

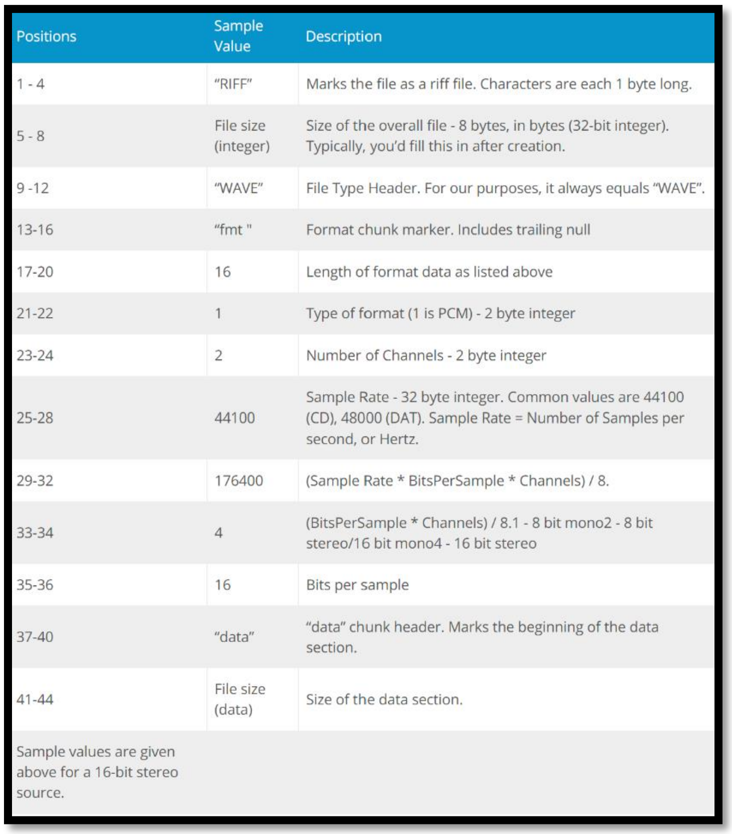
imageFileName – this is the relative path to the image file.

When the user gives the command of adding a song to the database, the first task of the server is to serialize the wave file. We have been mentioning wave files throughout the paper, but we still do not know what exactly this file is.

3.1.4. What is a wave file?

Waveform Audio File Format (WAVE or WAV) is a subset of Microsoft’s Resource Interchange File Format (RIFF) specification for storing digital audio files. This format does not apply compression to the bitstream, and it stores the audio with different sampling rates and bitrates (“Wave File Specifications”). Having no compression means that these files are widely supported and can provide very high audio quality. The maximum size of a wave file is 4GB because of its 32-bit unsigned integer that specifies the size of the file. Working with wave files is quite easy and they are widely supported.

A Wave file consists of two parts – header and data. The header stores very important information describing the properties of each wave file. The header is 44 bytes long and it has the following format:



Positions	Sample Value	Description
1 - 4	“RIFF”	Marks the file as a riff file. Characters are each 1 byte long.
5 - 8	File size (integer)	Size of the overall file - 8 bytes, in bytes (32-bit integer). Typically, you’d fill this in after creation.
9 -12	“WAVE”	File Type Header. For our purposes, it always equals “WAVE”.
13-16	“fmt "	Format chunk marker. Includes trailing null
17-20	16	Length of format data as listed above
21-22	1	Type of format (1 is PCM) - 2 byte integer
23-24	2	Number of Channels - 2 byte integer
25-28	44100	Sample Rate - 32 byte integer. Common values are 44100 (CD), 48000 (DAT). Sample Rate = Number of Samples per second, or Hertz.
29-32	176400	(Sample Rate * BitsPerSample * Channels) / 8.
33-34	4	(BitsPerSample * Channels) / 8.1 - 8 bit mono2 - 8 bit stereo/16 bit mono4 - 16 bit stereo
35-36	16	Bits per sample
37-40	“data”	“data” chunk header. Marks the beginning of the data section.
41-44	File size (data)	Size of the data section.
Sample values are given above for a 16-bit stereo source.		

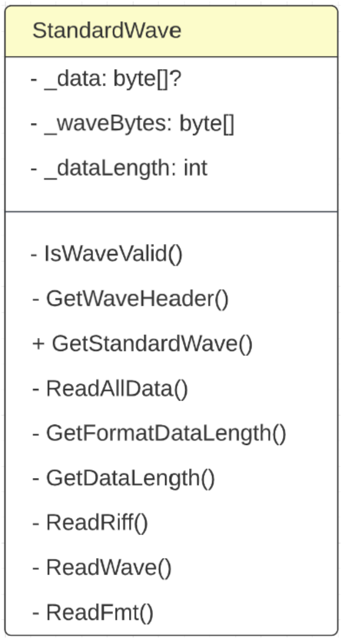
docs.fileformat.com

Some wave files also have additional information in the header, but our Client and Server application will not consider those. All the essential data to play wave files is the one described above. The extra information is after Bits per sample value and before the data chunk header.

The first step to stream audio from the server application and the client application is to standardize the wave files. The server takes any wave file, and it applies numerous functions to extract the necessary information from the header and the data, leaving extra information. Thereafter, the essential header data is concatenated with the header chunk header and data file size creating a standardized wave file. This file is then used by the server and client application. As an agreement, the client will only recognize and play standardized wave files that are supplied by the server.

The server contains a dedicated component that performs all those functionalities – StandardWave Class.

The UML class diagram for StandardWave is the following:

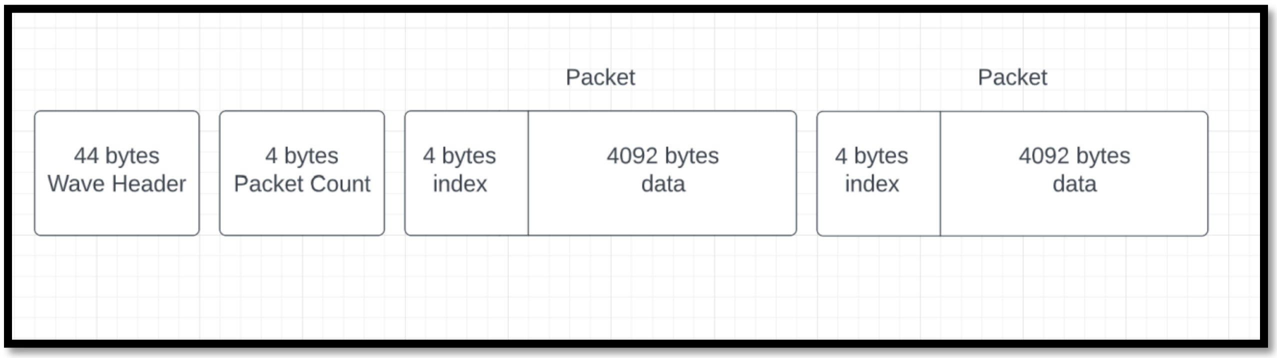


The class contains only one public method GetStandardWave(), which returns the standard wave file without any extra information. This class is not owned by any other classes. Each time the server needs to standardize a wave file, it will create an instance of this class, and it will call GetStandardWave().

Furthermore, the server application uses the AudioFile static class to save the standardized wave file using a very specific protocol that only our server application and the client application can understand.

3.1.5. AudioFile Static Class

This class only has static methods that perform various manipulation on the standardized wave file so that it can efficiently and correctly be used by our server and client applications. The public method GetAudioBytes(), with the help of the other private static methods will transform the wave file in the following format:



The format of wave files saved in the Server

The AudioFile Class does the following:

1) Get the wave header from a standardized wave file.
2) Take the data from a standardized wave file.
3) Split the data into packets of size 4092 bytes, and before each packet add the index at which this packet begins.
4) Reconstruct the file into a byte's array of the format from the diagram.

When this process is done, the server saves this byte array into byte's file in its storage.

3.2. Client Side

.NET does not provide many libraries to play audio files. Although wave files are supported by virtually every media library, .NET does not allow low-level interaction with how you play files. Two decent alternatives to play wave files in .NET are Soundplayer and MediaPlayer. Although they can play audio efficiently, they do not provide low-level control which is needed for an audio streaming service. The most crucial element of playing music is to have low-level control of the stream. For this project, I wanted to add numerous features which will make playing files very convenient and efficient. The service will be using the TCP protocol, and as such, the data will be supplied over time. Since wave files are uncompressed, their size is relatively large. A four-minute song stored in wave format is around 40 MB. Libraries such as Soundplayer and MediaPlayer can only start playing a wave song when all the data is available. But, as I previously mentioned, the goal of this project is to build an audio streaming service that is convenient, responsive, and efficient. If we were to use the .NET libraries, the application would not be responsive. For instance, the user asks for a song; the server finds the song and sends it back to the user; the user clicks play. In this situation, after the user clicks play, the server will receive the command and it will start sending the song which might take a couple of seconds (could take more than 40 seconds in slow connections). This would not be a good experience for the user. When the user clicks play, they expect the song to start immediately. Due to the above reasons, I decided to build my own audio library with C#.

The client side is the most complex part of the audio streaming service. The Visual Studio Project contains three folders that contain different parts of the application. The first component of the client is the Wave Out Wrapper – it is found in the WaveOut folder. The wrapper contains two classes – WaveOutBuffer, and WaveOutPlayer, and one C# file WaveNative that contains native functions and some necessary classes such as WaveFormat and WaveHeader.

3.2.1. WaveOut

We will start the description of the Wave Out Wrapper with the WaveOutBuffer. Wave Out was designed to work with audio buffers. We establish a connection with the physical speakers on our computer, and we send audio buffers that can be played by the speakers. Each buffer has a pointer to the waveOutHandle, a size, and an ID. On creation, they are prepared and ready to be played by the WaveOut device. Each buffer owns a pointer to some wave data. Using its size and this pointer we can update the data each time we need to fill the buffer with new data. As you can probably tell, the buffer is not very intelligent. In fact, it was designed this way because the buffer is not concerned with any functionalities but to carry data around. It is the same case with classes such as WaveHeader, WindowsNative, and WaveFormat. WaveHeader and WaveFormat only contain data, whereas WaveNative only contains native Windows Functions.

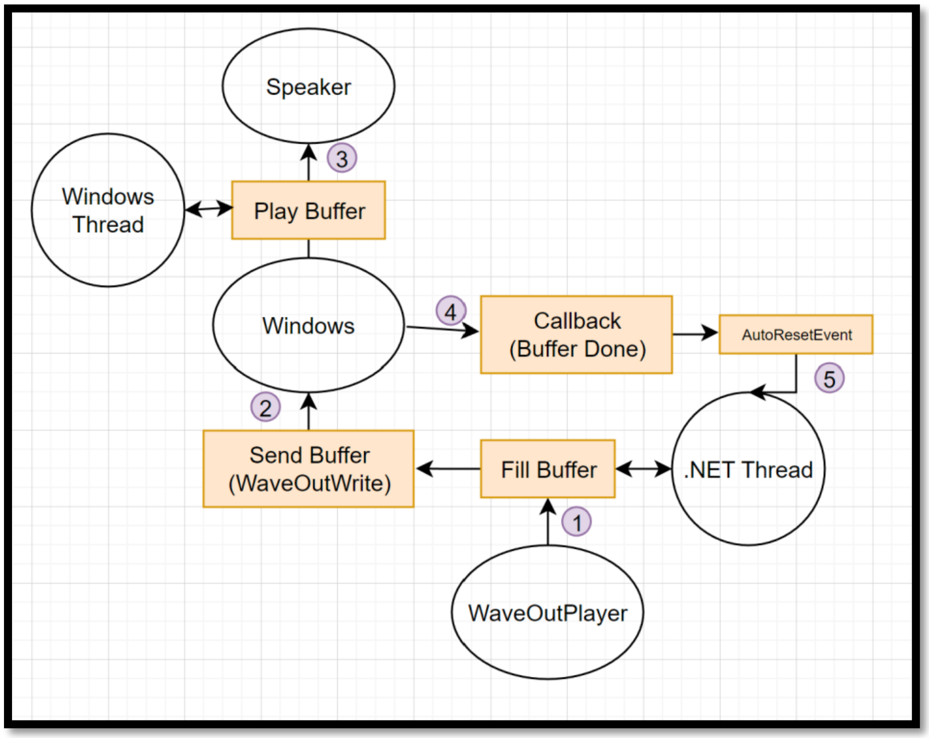
On the contrary, WaveOutPlayer class, which is part of the Wave Out Wrapper, contains many functionalities concerned with playing audio. This class is the heart of the application because, without it, the application cannot perform its main function – playing sound. This component takes care of the buffers and handles callbacks from Windows. WaveOutPlayer owns all the WaveOutBuffers in the application. This component is responsible for all communication between our client application and the speakers in the computer via the Windows native methods. If something goes wrong here, it is likely that the application will crash and stop playing audio. Thus, it was a high priority to design this class to be as consistent as possible. There are numerous scenarios that I, as the developer of this project, do not have control over. When we decide to use native Windows calls, we need to be careful of possible deadlocks and race conditions that might happen. Although I extensively used Microsoft's documentation on Wave Out, I had many cases when the application was having bugs, errors, and possible deadlocks. It is important to know that when using Wave Out we do not have much control over how and when the data is played. We simply supply some data and wait for a response that the buffer has been played.

The first step in playing audio is to open the connection with the waveOutOpen() native function. Here we tell Windows that we are trying to open a waveform-audio output device for playback. We supply the device we want to open (-1 for default), the format of the wave file, a handle for the device, and a callback function. The callback function is one of the most important components of the WaveOutPlayer. This is how Windows tells us that the buffer we sent has been played and is now being returned. After the buffer has been returned, we can reuse it – fill it with new data and send it again to be played. We can play a buffer using waveOutWrite().

The device will call the callback function passing a handle to the waveform-audio device associated with the callback, a waveform-audio message (WOM_CLOSE, WOM_DONE, or WOM_OPEN), a user-instance data, a WaveHeader, and an optional parameter. The WaveHeader is what we use to determine what exactly has been returned. The WaveHeader is a struct containing all the information about the buffer such as the pointer to its data, its length, dwInstance, flags, etc. From there, we use dwInstance, which in our case represents the ID of the buffer. Using Marshal.ReadInt32(), we can read what is stored in the pointer dwInstance ("Marshal Class (System.Runtime.InteropServices)"). Based on the ID that is returned, we can play the next buffer.

As you can tell, just by calling waveOutOpen(), Windows has created a separate external thread where it plays audio. We do not have any control over this thread. According to Microsoft, we should not use any native methods inside the callback function – we cannot simply add waveOutWrite() inside the callback function and hope for Windows to create a circular playback. In fact, in one of my first attempts, I implemented a player which is entirely handled by Windows, but there were many bugs, errors, and internal exceptions that cannot be caught, possibly caused by deadlocks. I fixed this issue by using a separate thread which is managed by a WaitHandle (more specifically AutoResetEvent). An Auto Reset Event is a thread synchronization event that when signaled allows a waiting thread to proceed, and it automatically resets to blocking mode, blocking the next thread. The Auto Reset Event is simply telling us that a callback function from Windows has happened and is now time to reuse the buffer.

This is a diagram showing how the Wave Out Player manages to play Audio:



WaveOutPlayer interacting with Windows

The diagram starts with WaveOutPlayer. In this simple diagram, we have only included one buffer, but realistically, there are at least 2 or more buffers. WaveOutPlayer will fill a buffer with data and send it for playback. Windows will take the buffer and play it. Thereafter, Windows will notify the player that the buffer has been played using the callback. This callback contains identification information about the buffer. The callback will simply enqueue the buffer for filling in the WaveOutPlayer and signal the thread which is responsible for refilling and playing the buffer again. This is how we can play audio using Wave Out. Now, we have significant control over how we play audio, which is essential for our audio streaming service. We simply need to supply some data to the buffers, and they will start playing sound.

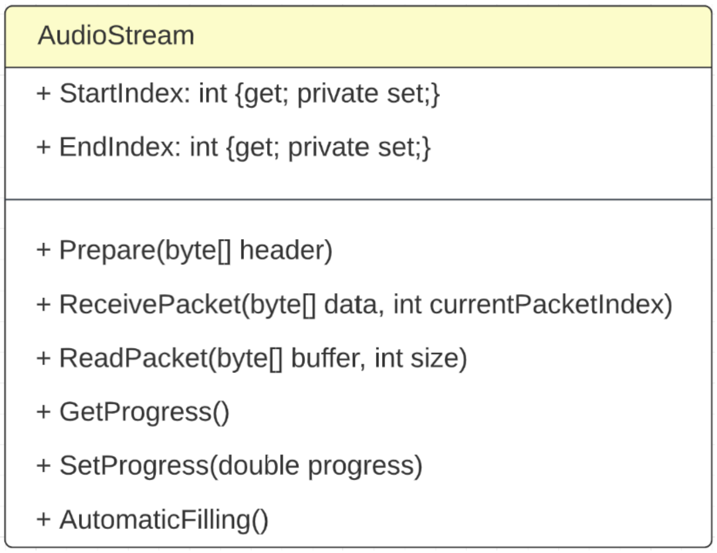
But how does the player know what data to read next?

This is managed by another essential component of our design – AudioStream Class. The AudioStream component is where the player will take all its data. This Class can be manipulated by multiple threads, and it is synchronized using locks and Reset Events in .Net.

3.2.2. AudioStream Class

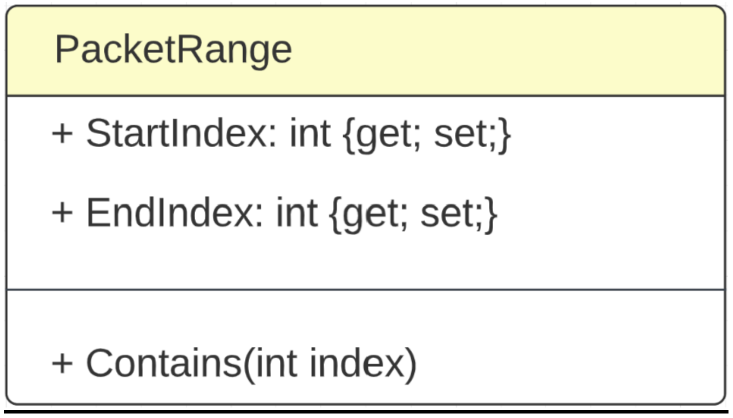
Using the WaveOutPlayer and WaveOutBuffer, we have low-level control over how the audio is played. We can have a stream or array containing data and using the filler passed to WaveOutPlayer, we can supply data to the buffers which will play the wave audio. To provide very fast playback over a TCP connection we need to receive data from the server and immediately start playing when we have enough data. Based on how fast we are receiving data we can adjust the behavior of the client with the help of the custom audio stream. AudioStream allows simultaneous reads and writes to its data. Using a network stream, we can supply data to the audio stream, and using the filler of the playing device we can take data from the stream. Moreover, the stream can keep track of how much data has been written and read, and where those operations have been done. The audio streaming service supports dynamic playback. This means that the user can request any part of the audio at any time, and it will be played without delay as if it was fully transmitted. This is especially necessary when the user plays long audio files such as podcasts and long songs. This is done by keeping track of how much data the AudioStream has received, and where this data has been received. For instance, if the user starts playing a four-minute song and decides to skip to the middle of the song immediately after they clicked play, the AudioStream will not wait for the data to become available – instead, it will check what data it has, and what data is being requested. If the data requested is not expected to become available soon, then AudioStream will communicate to the application that it needs data at the specified location, and the application sends a request to the server to send the specified packets.

The UML class diagram including only public properties and public methods for the Audio Stream Class is the following:


AudioStream
+ StartIndex: int {get; private set;}
+ EndIndex: int {get; private set;}
+ Prepare(byte[] header)
+ ReceivePacket(byte[] data, int currentPacketIndex)
+ ReadPacket(byte[] buffer, int size)
+ GetProgress()
+ SetProgress(double progress)
+ AutomaticFilling()

AudioStream Class Diagram

Now, I will describe in detail how AudioStream manages to give our client application so many features related to smooth playback. First, the AudioStream class utilizes a class called PacketRange. PacketRange is a simple class I created that has the following UML class diagram:



PacketRange Class Diagram

This class has two public properties – StartIndex and EndIndex. StartIndex specifies at which position in the entire data part of the wave file this packet range starts. EndIndex specifies at which position in the entire data part of the wave file this packet range end. The Contains() method checks whether the packet range has a specific index. The Audio Stream owns a list of Packet Ranges, and when it receives more data from the streaming socket, it updates the ranges accordingly.

<u>3.2.2.1. ReceivePacket() in AudioStream</u>

This method is called by a higher-level class – MediaPlayer class – that owns the AudioStream. From the UML diagram of the AudioStream Class, you can see that this method expects an array of bytes and the current packet index. The array of bytes that is supplied to this method is the packet itself. The constructor of the AudioStream class expects the size of the packet and the buffer size of the player. Calling ReceivePacket() is how we supply wave data to the AudioStream. The data is stored in an array, whereas the range is stored in the packet ranges list. When this method receives data, it will internally update the list of packet ranges that it owns. This is done in the following way:

1) Receive the index of the current packet that was supplied.
2) For any packet range, check whether there is a range whose End Index is equal to the current packet index.
    a. If there is such a packet, set the End Index of the packet to the current index, therefore making the range longer by packet size amount.
        i. If the index was added successfully to the packet range, check if the packet range on the right has a Start Index equal to the End Index of the current packet. If that is the case concatenate the packet ranges into one by setting the End Index of the current packet to the End Index of the packet on the right. Finally, delete the packet on the right.
    b. If there is no such a range, add a new range that starts at the current packet index and ends at the current packet index plus packet size.

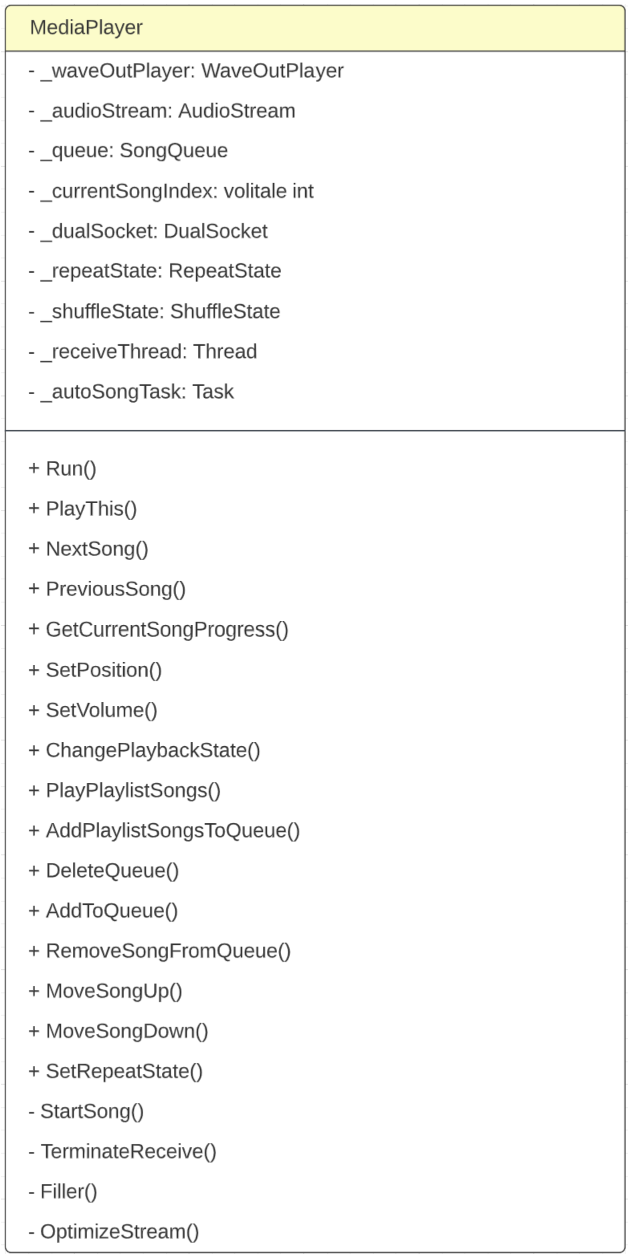
By using this algorithm, we know exactly what data we have, and where this data is located (Start Index and End Index). If there is more than one range, it means that we do not have all the packets, and the stream is not complete. If we have only one range that starts and 0 and ends at data size, then we have all the data.

3.2.2.2. ReadPacket() in AudioStream

While ReceivePacket() is used to supply data to the AudioStream, ReadPacket() is used to read data from the stream. This method can be called by at most one higher-level class. In our case, this higher-level class is the MediaPlayer Class that owns the AudioStream. ReadPacket() utilizes the ReadIndex property to determine where it needs to read data next. The MediaPlayer merely tells the AudioStream that it needs to fill a specific buffer, and the AudioStream does everything internally - there is low coupling between these components. Using packet ranges, the AudioStream knows whether it has data at the current ReadIndex. The caller is notified whether there is data at the ReadIndex. The AudioStream is synchronized with locks to support simultaneous reads and writes. Next, we will describe the design of the MediaPlayer Class and continue with Dynamic Streaming.

3.2.3. MediaPlayer Class

This is a high-level class that supports all the basic functionalities of a media player and more. This class is owned by the ControllerClass. MediaPlayer controls playback functions and the queue of songs. It owns both, the audio stream, and the wave/ out player. It also owns the streaming socket and is responsible for directly communicating with the server via this socket. This socket will be used to supply wave data to the media player. The MediaPlayer gives the responsibility of saving and understanding this data to the AudioStream. The MediaPlayer is also responsible for establishing the connection between the AudioStream and the WaveOutPlayer. The following is the UML class diagram for MediaPlayer:



MediaPlayer Class Diagram

As you can see from the diagram, MediaPlayer can perform many functionalities. One of the more important functionalities of the MediaPlayer is to make communication between the AudioStream and WaveOutPlayer possible. AudioStream and WaveOutPlayer do not know anything about each other – their functionalities are separated. WaveOutPlayer's task is to play sound using the audio devices in the computer, and it requests wave data when its current buffer is ready. AudioStream, on the other hand, accepts to receive data, and it can provide data to external sources that need it by calling its ReadPacket() method. It is the MediaPlayer that synchronizes these components together. MediaPlayer also has additional functionalities, as you can see from its private and public methods.

First, the MediaPlayer owns a DualSocket object, and it can use its internal socket to receive wave data from the server. This process is usually initiated after calling the private method StartSong().

3.2.3.1. StartSong() in MediaPlayer

This method is the lower-level method that will start playing the song at the current index from the queue. This method does the following:

1) Stop the WaveOutPlayer in case it is playing.
2) Terminate the receiving of data from the server in case it is receiving data.
3) Request a song from the server by sending the song ID of the song at the current queue index.
4) Send the mode of streaming – MOD1 – represents that it is a normal streaming request.
5) Tell the streaming thread that it should start receiving data from the server.
6) Immediately start the WaveOutPlayer

The MediaPlayer can also start receiving data even without calling the StartSong() method. This is done when the MediaPlayer needs to initiate Dynamic Streaming. Dynamic Streaming is handled by the AudioStream, but only the MediaPlayer can initiate it. When the MediaPlayer receives a request from AudioStream to start Dynamic Streaming, it will terminate the process of receiving data in MOD1. Afterward, it will start receiving wave data again by calling its StartReceiveData() method.

3.2.4. Dynamic Streaming

Dynamic Streaming is the feature that utilizes many functionalities of our audio stream and the client application in general. This feature is activated only in specific scenarios. It provides our client application with the ability to provide a smooth playback experience to the user. Although our audio streaming application is using wave files for maximum audio quality, the audio streaming service reacts as if the size of such files is irrelevant.

Dynamic Streaming is only activated if the user changes the progress of the wave file. The Dynamic Streaming algorithm is interesting because it runs over the network. This algorithm consists of two parts. The first part is run in the client application, whereas the second part is run in the server application. For the client side we have the following:

1) User changes index
    a. If WriteIndex is further than 300 buffers away on the left:
        i. The WriteIndex is too far on the left. Request to initiate dynamic streaming and fill the space between ReadIndex and the end of the stream. This way we will have all the data on the right, where the WaveOutPlayer will request data from.
    b. Else if WriteIndex is closer than 300 buffers away on the left:
        i. If the AudioStream does not have packets from ReadIndex until the end of the stream start receiving from ReadIndex until the end of the stream.
        ii. Else if it has packets from ReadIndex until the end, the client application simply waits for more buffers – this will take a relatively short amount of time.
    c. Else if WriteIndex is bigger than ReadIndex:
        i. If the AudioStream does not have all the packets from ReadIndex until WriteIndex it should initiate dynamic streaming to fill the space between ReadIndex and WriteIndex.
2) If Dynamic Streaming is initiated, the AudioStream will ask the MediaPlayer for specific packets – from index A to index B.

To save resources, dynamic streaming optimization uses the same streaming socket as standard streaming. The difference is that it starts the streaming request with MOD2. The MediaPlayer, after it receives the request from AudioStream via a callback function, does the following:

1) It sends a terminate stream request to the server so that the server does not send data that is not needed now and is ready to send data that we need currently.
2) It sends to the sever the mode of streaming request – in this case MOD2.
3) It sends the index of the beginning of the range.
4) It sends the index of the end of the range.
5) It prepares the streaming loop to receive data from the server.

The algorithm of the client side does not end here, but the next step is in the server application.

After the client sends the mode of the streaming request, the server client handler decides what to do. If the mode of streaming is MOD2, it will start sending specific ranges of wave data. When the client user clicks play on a specific song, the first mode of streaming is always MOD1. At the beginning of MOD1, the server saves a copy of the entire wave file so that it can use it in case the client requests MOD2. In MOD2, we have the start index of the range and the end index of the range. The server will adjust the indexes of the beginning and end of the range so that they match the specific indexes in the wave file. When we were looking into the design of the Server Application and its AudioFile static class, we mentioned that the modified format of the wave file contains not only the data but also the index at which a packet of 4092 bytes of data starts. The server application uses this fact to determine where exactly in the wave file it needs to retrieve data from. Once this range of data is retrieved, it will start sending it to the client application by specifying MOD2.

Back to the client-side part of the dynamic streaming algorithm, the client will receive the data at the index it requested until the end index that it requested. When and if all of this ranged data is received, the ReceiveFromServer() method of MediaPlayer Class which is running in a separate Thread and is responsible for all of the wave file streaming, will notify the AudioStream that initiated the dynamic streaming request in the first place. This is done in the AutomaticFilling() method of the AudioStream.

3.2.5. AutomaticFilling() in AudioStream

This method is called each time the MediaPlayer finishes streaming in MOD2. This is done so that the locations where the client application did not manage to reach, are filled with data in case the user decides to go back to this location. This is where the algorithm finishes, but as you will see, the algorithm continues until all the data is received. In AutomaticFilling() we have the following steps:

1) Check whether the AudioStream has all data from index 0 until the end of the stream.
    a. If it has all data, do nothing.
    b. If it does not have all data, check the number of ranges AudioStream has.
        i. If there is only one range it means that the range starts at 0, but its EndIndex is not the end of the stream. Therefore, we need to start dynamic streaming requesting packets from the EndIndex until the end of the stream.
        ii. If there is more than one range, it means that there is a least one empty space in the wave data.
        iii. If there is an empty space on the right of the ReadIndex, dynamic streaming should prioritize filling the right side.
        iv. If there is no empty space on the right of the ReadIndex, we can start filling it from the beginning.

2) Start Dynamic Streaming based on the previous conditions.

Now that dynamic streaming is restarted, the process will go into a loop passing data via the network until all the wave data is received from the client. Thus, we have managed to provide smooth playback without compromising the performance of our applications and the network.

4. Implementation

For the development of this senior project, I have used Visual Studio 2022 IDE with .NET 6. I have used the WPF Application template for both client and server applications.

.NET 6 supports numerous types of applications, and you can build virtually any type of program with it. In my case, I wanted to focus more on the lower-level components of the client and server applications and worry less about the graphical user interface (.“NET (and .NET Core) - Introduction and Overview”). Therefore, using the WPF Application template utilized in .NET 6 gives me the option to have all the necessary components for the development of a Client-Server architecture and a convenient and simple way of building a graphical interface.

The programming language of my choice for this project is C#. I have substantial experience with C#, and I feel confident in my skills in developing desktop applications using .NET and C#. C# is a modern language that is designed to be effective and fast. It is great for developing object-oriented projects that are easy to maintain. This programming language is quite popular and well-documented. By going to the official page of Microsoft, we can read the documentation of any components that are designed for C#. Visual Studio IDE also works great with C#. Debugging is very easy to do, and in the recent versions of Visual Studio IDE, Microsoft has added AI to make the process of writing code more efficient. Furthermore, .NET and C# have great implementations of Threads, which are essential for client and server applications. Multiple components of the client and server applications are implemented by using threads.

4.1. Multithreading

For this specific project, parallelism is very important. In the client application, each of the main entities such as WaveOutPlayer, MediaPlayer, and Controller use .Net threads (“Using Threads and Threading”). The same applies to the server application – the Controller and ClientHandler both use .Net threads. To make the applications responsive and in some cases functional, having parallel tasks is essential.

4.1.1. WaveOutPlayer Multithreading

WaveOutPlayer directly communicates with “winmm.dll” by allocating buffers, preparing them, filling them, and finally sending them. This is a very fast process which is highly dependent on the performance of the CPU in the computer. The faster the CPU, the smaller the buffer size can be. The smaller the buffer, the less latency there is. In other word, if we want low latency smooth and high-quality audio, we need a dedicated thread to perform this task as fast as possible. C# threads, unlike some other programming languages such as Python, can be run in separate cores which increases the performance of the application.

4.1.2. MediaPlayer Multithreading

MediaPlayer, apart from functions such as queueing songs, skipping songs, and removing songs from the queue, has the very important task of controlling the flow of the audio data that is being received and how it is sent to the AudioStream. It is important to have a dedicated thread that reads the data from the streaming socket and sends it to the AudioStream. This function will be under consistent load for a relatively long time, depending on the speed of the connection. This will cause the MediaPlayer to block if reading is not performed in a different thread. Even if the UI is responsive, when the user clicks next, previous, or anything else concerning the MediaPlayer, the commands will not be executed until the MediaPlayer has received all the audio data from the server. Multithreading solves this problem.

4.1.3. Client Controller Multithreading

In the client application, the controller must be responsive all the time. If the controller is not responding, the communication between the client and the server will be interrupted. This will affect all the communication requests such as searching for a song or terminating streaming. Furthermore, the controller is responsible for handling commands sent by the user such as inputs, buttons, etc. This component has many tasks that should not block one another. Therefore, by using .Net threads and tasks we can provide parallelism in the controller too.

4.1.4. Server Controller Multithreading

When implementing the server side, the need for using threads was even higher. To support multiple clients at the same time which can be doing simultaneous requests it is essential to have concurrent or parallel execution. I have used two threads that are owned by the controller that accept requests on specific ports. One port is reserved for streaming, and the other port is reserved for communication. The use of threads can be substituted with .Net Tasks since accepting requests and assigning client handlers do not require a lot of processing power.

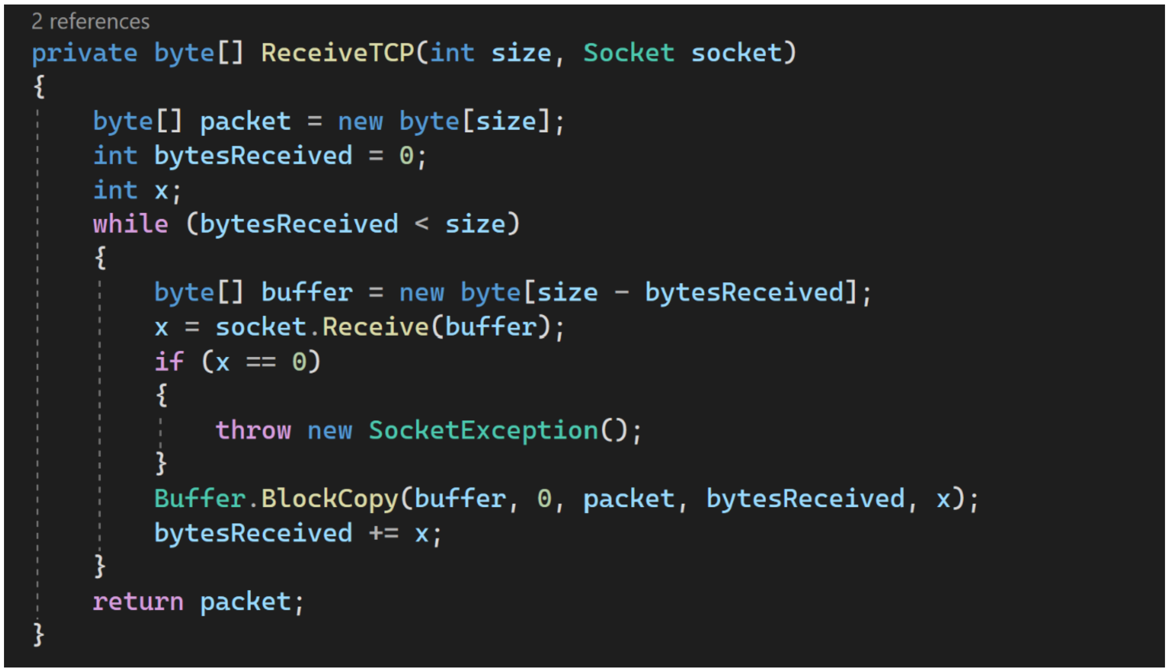
4.1.5 ClientHandler Multithreading

In the client handler, by using two separate threads – one for the communication and one for streaming – we achieve parallelism between two processes that need to be responsive and ready to change. The streaming thread will be under load for an extensive period, and the communication thread is important for communication between the server and the client. The communication thread also can state the change of the streaming thread in if the client sends a termination request or a search song request.

Next, we will continue with the implementation of the TCP protocol and the choices we have taken.

4.2. .NET TCP Modifications

Although TCP is a reliable protocol, and we are guaranteed to receive the bytes we sent, sometimes Socket.Send(byte[] buffer) will not send the entire buffer, and Socket.Receive(byte[] buffer) will not receive the entire buffer when called. To solve this issue, I have written two functions that will send and receive the entire buffer – ReceiveTCP() and SendTCP().

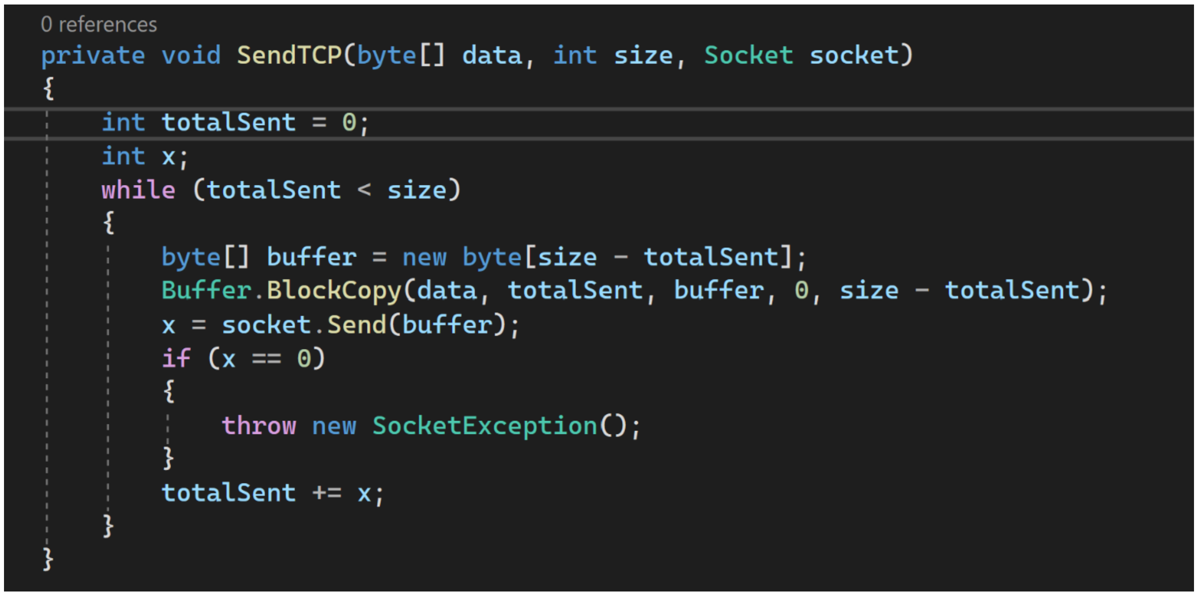


```
2 references
private byte[] ReceiveTCP(int size, Socket socket)
{
    byte[] packet = new byte[size];
    int bytesReceived = 0;
    int x;
    while (bytesReceived < size)
    {
        byte[] buffer = new byte[size - bytesReceived];
        x = socket.Receive(buffer);
        if (x == 0)
        {
            throw new SocketException();
        }
        Buffer.BlockCopy(buffer, 0, packet, bytesReceived, x);
        bytesReceived += x;
    }
    return packet;
}
```

Receiving the Entire Buffer

ReceiveTCP() takes two arguments – int size and Socket socket. The full buffer is declared as byte[] packet = new byte[size]. This is the packet we are expecting to receive. The while loop will check whether we have received as many bytes as size. byte[] buffer will store the current buffer that is being received. The socket will receive the bytes and store them in the buffer. The bytes will be copied into the packet. x specifies the total number of bytes that were received. If the received bytes were not as many as we expected, the code inside the while loop will be run again. The data from the socket will be stored in the buffer again, and the remaining data will be appended to the packet. If x is 0, according to Microsoft's Documentation on Sockets, it means that

Socket.ShutDown() has been called, or a disconnection happened. We throw an exception, which is later handled.



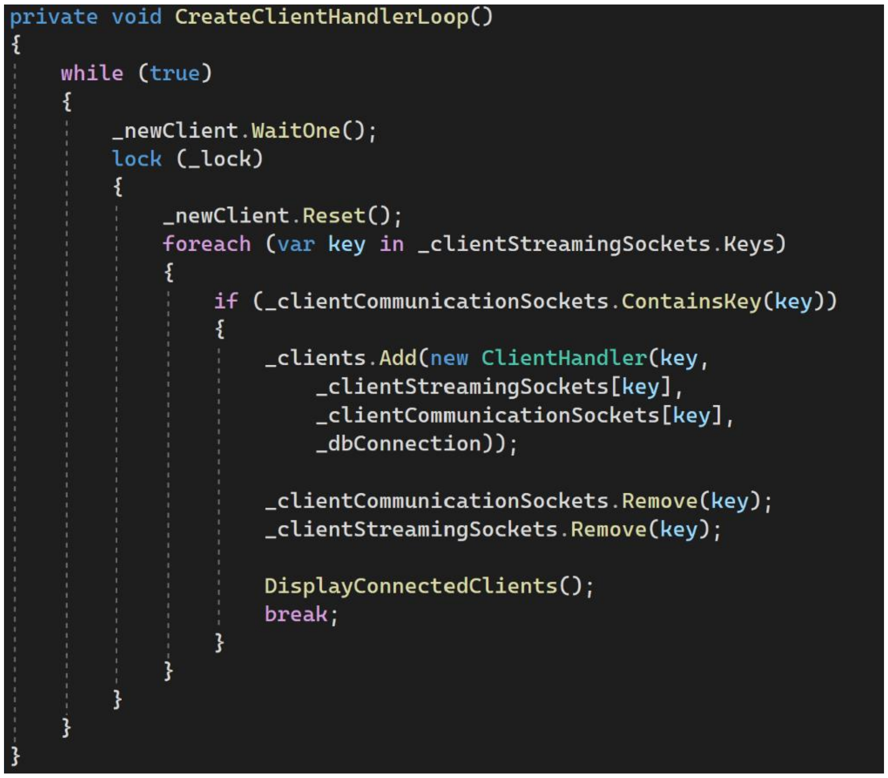
```
0 references
private void SendTCP(byte[] data, int size, Socket socket)
{
    int totalSent = 0;
    int x;
    while (totalSent < size)
    {
        byte[] buffer = new byte[size - totalSent];
        Buffer.BlockCopy(data, totalSent, buffer, 0, size - totalSent);
        x = socket.Send(buffer);
        if (x == 0)
        {
            throw new SocketException();
        }
        totalSent += x;
    }
}
```

Sending the Entire Buffer

Similarly, Socket.Receive(), Socket.Send() may not send all the data at once. We keep track of how much data has been sent. If this amount is less than the size we specified in the beginning, we try to send again the remaining data. If the amount of data sent was 0, it means that the socket has been disconnected and an error is thrown.

4.3. Dual Socket Implementation

Furthermore, in both applications I have implemented a dual socket design. We have mentioned throughout this paper that there are two parallel sockets providing streaming and communication between the client and the server. These sockets, although perform different task – one sends and receives wave data, while the other sends and receives communication messages – they are closely related. First, to establish a full connection with a client, the server needs two sockets running in different ports and that are connected to the same client. As we previously mentioned, there are two separate threads accepting requests in different ports. If the communication listening thread accepts a communication socket, it will check if there is a streaming socket that matches the client. On the other hand, if the streaming listening thread accepts a streaming socket, it will check if there is a communication socket that matches the client. This is implemented in the CreateClientHandlerLoop() below:



```
private void CreateClientHandlerLoop()
{
    while (true)
    {
        _newClient.WaitOne();
        lock (_lock)
        {
            _newClient.Reset();
            foreach (var key in _clientStreamingSockets.Keys)
            {
                if (_clientCommunicationSockets.ContainsKey(key))
                {
                    _clients.Add(new ClientHandler(key,
                        _clientStreamingSockets[key],
                        _clientCommunicationSockets[key],
                        _dbConnection));

                    _clientCommunicationSockets.Remove(key);
                    _clientStreamingSockets.Remove(key);

                    DisplayConnectedClients();
                    break;
                }
            }
        }
    }
}
```

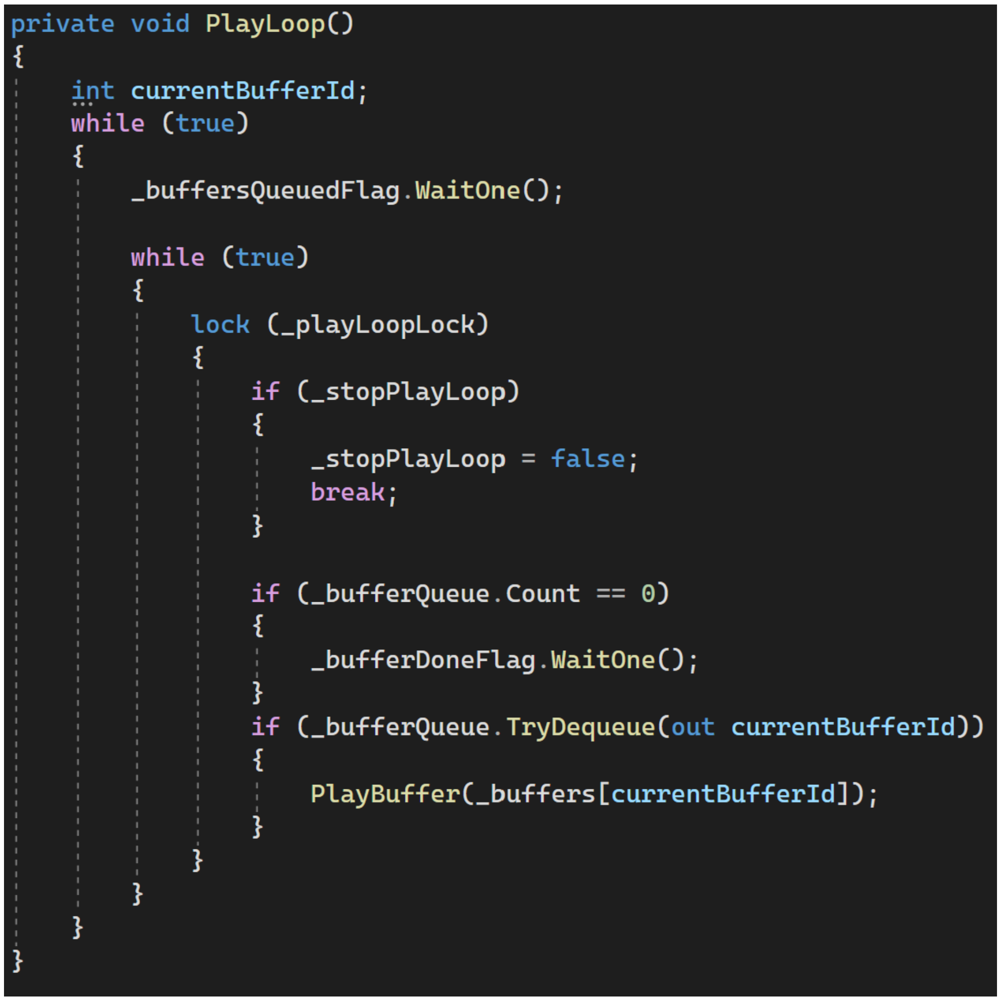
Creating a Client Handler in the Server

This method is run by a .Net task. It is a lightweight function that checks if there are two sockets of different types connected to the same client. The while loop waits at _newClient.WaitOne(). This is a ManualResetEvent that needs to be signaled before letting the task proceed. This signal comes from the communication listening thread and a streaming listening thread. As you can see, there is a lock right after _newCllient.WaitOne(). The same lock is used when the streaming and communication listening threads add a new client to the _clientStreamingSockets and _clientCommunicationSockets dictionaries, respectively. This is done to prevent the parallel threads from causing errors in the foreach loop by adding new clients while the foreach loop is running. Once the CreateClientHandlerLoop() finds two different sockets connected to the same client, it will create a new ClientHandler object that handles client requests.

I have implemented the dual-socket design in the client too. In fact, there is a dedicated class called DualSocket that owns one streaming socket and one communication socket. I have implemented it this way because this is a very efficient way of meeting the functional requirement of reconnection in case the client disconnects. As we previously mentioned, the server will only create a ClientHandler if the streaming and communication sockets are both available. We can determine whether the client has disconnected by catching SocketExceptions. When this happens, we can call the Reconnect() method of the DualSocket. This method will make sure that in case either of the sockets is disconnected, there will always be two new sockets assigned for streaming and communication.

4.4. Implementing the WaveOutPlayer

With the help of the documentation provided by Microsoft on wave/ out, I managed to implement the player of our client application – the WaveOutPlayer. wave/ out is part of Windows Multimedia, and it can be used by calling the native multimedia functions such as waveOutOpen, waveOutWrite, waveOutPrepare, etc. Throughout this paper, we have explained why having low-level control over playing sound is essential. The native functions have been imported from “winmm.dll”. You can find all the external functions in “WaveNative.cs” inside the “WaveOut” folder. There are two main components in WaveOutPlayer. The first component is the play loop; the second component is the wave/ out callback function. These two components work together to provide sound via the computer’s output devices. Below you can see the code for the play loop:



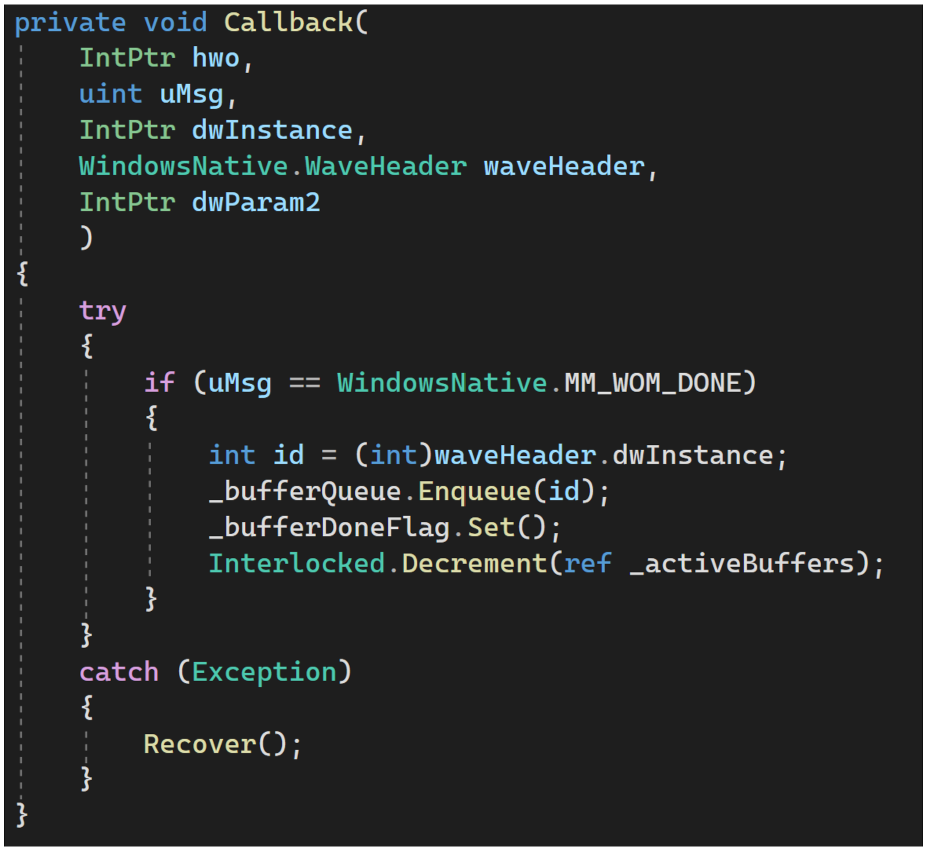
```
private void PlayLoop()
{
    int currentBufferId;
    while (true)
    {
        _buffersQueuedFlag.WaitOne();

        while (true)
        {
            lock (_playLoopLock)
            {
                if (_stopPlayLoop)
                {
                    _stopPlayLoop = false;
                    break;
                }

                if (_bufferQueue.Count == 0)
                {
                    _bufferDoneFlag.WaitOne();
                }
                if (_bufferQueue.TryDequeue(out currentBufferId))
                {
                    PlayBuffer(_buffers[currentBufferId]);
                }
            }
        }
    }
}
```

PlayLoop() in WaveOutPlayer

The PlayLoop() method runs in a separate thread managed by the WaveOutPlayer. This thread has its priority set to very high. This thread will be scheduled before any other threads. PlayLoop() contains two nested while loops. The innermost loop plays wave data from the current wave file. In case the application needs to change the wave file (when the user clicks the next song, previous song, etc.), PlayLoop() will break out of the innermost while loop. There are two AutoResetEvents (_buffersQueuedFlag and _bufferDoneFlag) that are synchronizing the thread. In the inner while loop, the AutoResetEvent will cause the thread to wait until the wave/ out has returned a buffer that is ready to be played. Once the thread is signaled, it will fill the buffer with data and send it back for playback. This loop will exit only when the client application stops the playback by setting _stopPlayLoop to true. Next, we will see the code for the wave/ out callback function:



```
private void Callback(
    IntPtr hwo,
    uint uMsg,
    IntPtr dwInstance,
    WindowsNative.WaveHeader waveHeader,
    IntPtr dwParam2
    )
{
    try
    {
        if (uMsg == WindowsNative.MM_WOM_DONE)
        {
            int id = (int)waveHeader.dwInstance;
            _bufferQueue.Enqueue(id);
            _bufferDoneFlag.Set();
            Interlocked.Decrement(ref _activeBuffers);
        }
    }
    catch (Exception)
    {
        Recover();
    }
}
```

Callback in WaveOutPlayer

Callback() is called by the external wave/ out thread every time Windows has played a buffer that was previously sent by us using waveOutWrite. There are two critical components in the signature of the callback – uMsg and dwInstance. The first one is used to determine what type of callback Windows is calling. In our case, we are interested only in callbacks that are sending buffers. The second one is used as an identifier for our buffers. This instance can be set by us, and I have chosen to use an integer number that is passed as a pointer. As you can see in the body of the function, every time Windows calls this function, we are appending the returned buffer to a queue that is used by the PlayLoop() to play the buffers. _bufferDoneFlag is signaled and all the threads that were waiting for it will proceed – in our case the PlayLoop() thread.

4.5. Sending Song Objects via TCP

When the user searches for a song, a request is sent to the server. The server will reply by specifying the number of songs that were found and the song themselves. As you could see in the database table that stores the songs in the server's database, there are multiple attributes such as song name, artist name, duration, an image in bytes, etc. All this data needs to be sent from the server to the client, and the client needs to understand it. SendTCP and ReceiveTCP only accept data in the form of bytes. To make the process of sending and receiving song data between the client and the server, I implemented a Song Class that has a serialization method and a deserialization constructor. The client and server only need to be using the same Song Class, and they will be able to easily send and receive song objects. First, the serialization method will take all the properties of the Song object and convert them into bytes. Second, it will calculate the size of each byte's property. Third, it will find the index at which the length of the property should be located. Fourth, it will append the lengths and the data into a byte array. The format is the following:

[Length] [Song ID] [Length] [Song Name] [Length] [Artist Name] etc.

To deserialize this object, the constructor of the Song Class simply needs to read the length amount of data in the byte array and convert it into the appropriate format. It needs to do this until it has read the last property.

To send the byte array via TCP is very easy – we simply slice the byte array into chunks and send the chunks to the client application. The client application takes all the chunks and concatenates them. When all the chunks are received, the client application simply passed the byte array into one of the Song Class constructors that will create a Song object.

Our client application also supports playlists. I have reused the serialization and deserialization functionality for playlists too. All playlists are saved locally. An individual playlist is represented by a folder in Windows. A byte’s file in the folder represents a song. We can use the serialization method of the Song Class to convert the song into a byte array and save the byte array into a byte file in the appropriate playlist folder.

4.6. External Libraries

SQLite for .Net is the only external library that I used in this project. All other libraries used come natively with .Net and Windows. SQLite is a library that implements a small, fast, self-contained, full-featured, SQL database engine (“About SQLite”). This library is used in the server application to save and retrieve song data. I have some experience with SQLite, and it is very easy to use. Since the server application will not perform many reads and writes to the database, SQLite is more than sufficient. By using this external library, I managed to build the database components of the server relatively quickly.

SQLite is included in the source files of the server application, so the user will not need to perform any additional steps. You can simply go to Sever Application > bin > Release/ Debug > net6.0 > Server Application.exe. The executable file will start the application and it will run on any Windows machine.

5. Testing

To test the server and client application, I used two laptops running Windows 11. The interaction between the client and the server has been tested in the same network over Wi-Fi. This will put realistic bandwidth limitations for the audio streaming service. The transfer speed between the computers averages around five megabytes per second or 40 megabits per second. To transfer a wave file from one computer to the other it takes a couple of seconds or minutes based on how large the file is. The bandwidth limitation is crucial for testing how well the audio streaming service performs in realistic scenarios.

5.1. Testing the Server Application

Before starting to test the server, we need to make sure that it is configured properly. By going to its app.config, we can see all the properties that need to be accurate before the server is started. The Host property needs to be changed based on where the application is being run. – this is the IP address of the server. The other do not need to change unless we want to. To get the IP address of the server, we open CMD and type ipconfig. From there we will take the IPv4 Address and set it as the value of the Host. This IPv4 Address needs to be used in all the clients that need to connect to this server too.

Now that we have properly configured the server application, we can start with testing. One of the functional requirements of the server is to be able to save songs in its database. The database folder is in Server Application > bin > Database. The Database folder contains two folders (Images and Songs) and a serverDatabase.db file. When we run the application, we are provided with the following:

1) Song Name: Text
2) Artist Name: Text
3) Open Song File: WAVE file
4) Open Image File: PNG file
5) Add to Database: Button

The user types the name of the song, and the name of the artist, chooses a wave file, chooses an image file, and clicks *Add to Database* button. The expected outcome is that we have a new byte file with the name {Song Name} by {Artist Name}.bytes in the Database/Songs folder and a {Song Name} by {Artist Name}.png in the Database/Images folder. By supping all these inputs and clicking on the Add to Database button, the server successfully saves the wave file, and image file, and updates the database file. We can go to the respective folders and check that all the data is there. Also, we can use DB Browser to check that the entry for the newly added song is there, and it indeed is.

The server application also manages to handle incorrect inputs. For instance, if the user forgets to give any of the required inputs, the server application will notify the user and there will be no changes to the database unless the user supplies the correct input. If the user forgets to supply the song name, a message that the song name is missing is displayed.

Furthermore, the server needs to support multiple clients at the same time. To test whether the server can handle two or more clients at the same time, we will need to run multiple client applications. First, we run the server at the appropriate IP Address. Second, we configure the client application to connect to this server. Third, we start two instances of the client application. The expected outcome is that two clients can start streaming audio at the same time, and the server can keep track of the connected clients. I have tested at most 10 clients streaming audio while connected to the server without any issues. The limitations of the server will be mainly set by the hardware of the computer that the server is running on.

Additionally, in case the client application disconnects or is terminated, the server releases any memory that was assigned to the client and closes the sockets. By using the Visual Studio Diagnostic Tools, we can see when the server releases memory. The server also removes the client from its main window when the client disconnects.

When it comes to the functional requirement of being able to send song objects and wave data to the client, it is not possible to set them in isolation. We will need to test these functionalities by using the client application itself. If the client asks for a song that we know is in the database, and the song is successfully sent and received by the server, it means that the server is performing its operations correctly. The same applies to wave files – if the client application user requests to play a specific song, that sound should start playing immediately. Therefore, the remaining part of testing the audio streaming service will be focused on the client application.

5.2. Testing the Client Application

One of the functional requirements of the client application is to connect to the server using the TCP protocol. If we have successfully implemented the TCP protocol in our audio streaming service, the client application will be able to do the following:

1) Automatically connect to the server after running the application
2) Search for a song or artist and receive matching songs
3) Click on the play button of any of the received songs and start playing sound

Regardless of where the client and server applications are running, if they are configured properly, they should be able to establish and maintain a dual TCP connection. When the applications are running on the same computer, we can supply the same IPv4 Address for both applications, and they should work. For my testing, to make it as realistic as possible, one of the computers ran the client application and the other ran the server application. The result when it comes to the TCP connection is that the client and the server applications manage to establish two connections via the streaming and communication sockets. When the user searches for a song, results are immediately sent. When the user asks to play a song, the song starts playing immediately.

Next, the client should be able to reconnect in case of disconnection. To test this requirement, we will need the server and client application to run on separate computers. The server application will be online for the entire duration of this testing. First, we check that the client and server are connected successfully by searching for a song and playing it. Second, we turn off the Wi-Fi of the client application machine. This will cause the client to disconnect from the network and ultimately from the server. Third, we turn the Wi-Fi back on. We expect the client application to reconnect automatically. This is indeed the case. By performing another search request and play request, the client starts playing sound.

Furthermore, the client application must be able to play wave files. This is achieved via the WaveOutPlayer. This component has been tested thoroughly during the development stage of the application. For our acceptance test, it is important that the client application plays clear sound without interruptions. Once we have established the connection with the server, we simply request to play a song. The expected outcome is that the output device in our machine starts playing sound. In my personal computer which has a CPU with 8 cores and 16 threads running at 4.9 GHz, the client application plays sound without any issues. The same applies to my other machine which has a CPU with 6 cores and 6 threads running at 4.0 GHz. I cannot confirm that the client application will be able to consistently play sound in machines with significantly weaker CPUs. I previously mentioned that wave/ out plays wave files using buffers. If these buffers are not supplied fast enough, the audio device will crash (Microsoft). However, even when I set the priority of the client application to efficiency mode in Task Manager, the application runs correctly. To make the testing more valid, I added a total of twenty songs to the queue for a total of one hour and twenty minutes of playback and let the application play sound until the end. The client application consistently passed the test.

When it comes to the functional requirement of supporting all the basic commands of a music player, I conducted numerous tests.

5.2.1. Play Song Command

When the user clicks on the play button on top of the song container, the client application is expected to play sound and update the UI. The application successfully starts playing the specified song, it updates the image container for the current song, it changes the song and artist name in the UI for the current song, it sets the progress of the bar to the beginning. Furthermore, if the client application was previously playing a song and the user clicks on the play button of a song, the previous song is stopped. Even when the user repeatedly presses the play button very fast, the application manages to handle the requests.

5.2.2. Play/ Pause Command

When the user clicks on the play/pause button the application is expected to pause the song if it was previously playing and resume the song if it was previously paused. This is indeed what happens. If the song was playing, the song will be stopped, the progress bar stops moving, the timers are paused, and the button is changed to indicate that the song is paused. If the song was paused, the song is resumed, the progress bar starts moving, the times are resumed, and the button is changed to indicate that the song is playing.

5.2.3. Previous Song Command

When the user clicks the previous song button, the application must play the song that was previously playing. First, we test the application with only one song in the queue. If there is only one song in the queue and the user clicks the previous song button, the same song starts playing since there is no other song to play. Second, we test the application by starting to play song A, and then we start playing song B. While we are at song B, we click the previous song button, and the player starts playing song A again. Also, if the user had set the controls to play song B on repeat, the state is changed since the user now requested to play the previous song. Furthermore, if the playback is paused and the user clicks on the previous song button, the playback is started at the previous song.

5.2.4. Next Song Command

When the user clicks the next song button, the application must play the next song from the queue. To test this requirement, first, we simply click on the play button of a song and do not add any other songs to the queue. After clicking the next song button, the client application is expected to keep playing the current song if there are no other songs in the queue. The client application passed this test. Afterward, I tested the next song command while there is a song in the queue. The client application manages to stop the current song, and it immediately after starts to play the next song from the queue.

5.2.5. Volume Up and Volume Down

To test this functionality, we simply need to check whether the volume changes according to the command. At 100% we must have the loudest sound, and at 0% we must have no sound. The volume control passed the following tests:

1) Sound is playing and we turn the volume down - the result is quitter sound.
2) Sound is playing and we turn the volume up – the result is louder sound.
3) No sound is playing, and we change the volume – the volume applies to songs afterwards.

5.2.6. Shuffling Command

When the user clicks the shuffle button, we expect to see the order of songs to change randomly. To test whether this functionality is met I did the following:

1) Added 10 songs in the queue.
2) Clicked the shuffle button.
3) The order of songs changed.
4) Repeat.

After many repetitions, I observed the order not only changed, but it was different most of the time. This means that the songs were successfully randomly ordered each time.

5.2.7. Un-shuffling Command

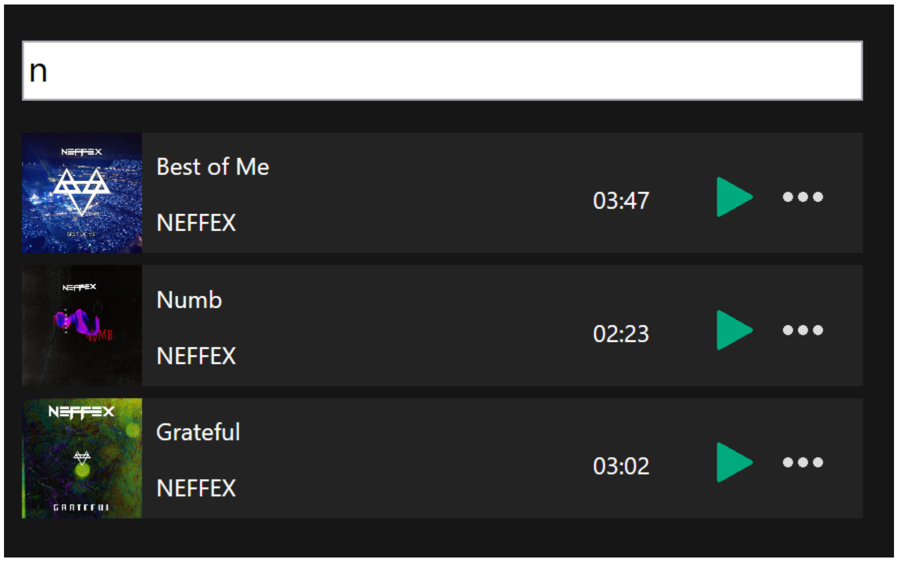
This command revers the changes done by the shuffling command. Testing this command is quite easy. We simply shuffle the queue by clicking the shuffle button. Next, we click the shuffle button again and expect all the changes to be reverted. With 10 songs added to the queue, this is exactly what happened.

5.2.8. Search for Song command

When the user starts entering some text in the search bar, the client application is expected to display songs that match the pattern. The pattern can match the song name or the artist's name. To test this functionality, we need to populate the server's database and storage with some songs. We need to know the exact name of the song and the artist for each song. During the testing I added four songs to the database:

1) Best of Me by NEFFEX
2) Numb by NEFFEX
3) Grateful by NEFFEX
4) Mortals by Warriyo

If we analyze the names of the songs and artists, we can see that the song Mortals by Warriyo does not contain the character 'n' and all the other three songs by NEFFEX contain this character in either song name, artist name, or both. Therefore, if the user types only the character 'n', we expect to see only the songs from NEFFEX. As you can see from the screenshot that I took, this is exactly what happens:



Searching for Songs

We can do a similar test with the character ‘w’. The only song that contains this character is Mortals by Warriyo in the artist’s name. When the user clicks on the play button of Best of Me, for example, Best of Me starts playing. When the user clicks on the more button of Grateful, an Add to Queue and Add to Playlist (if any) are displayed. When the user clicks on Add to queue, Grateful is successfully added to the queue.

5.2.9. Testing playlist creation, deletion, and renaming

To test these requirements, first, we need to go to Client Application > bin > Playlists. We need to make sure that there are no folders inside this folder. When the user starts the client application, there should be no playlists displayed in the playlist section on the left. When the user clicks on the plus button next to the “Playlists” label, a pop-up window should appear. Afterward, when the user enters a playlist name and confirms the operation, a new playlist should be created. We expect the following changes:

1) A new playlist must appear in the playlist section.
2) A new empty folder must be created inside the “Playlists” folder.

To test the deletion of a playlist, we need to click on one of the playlists (if any) and then click on the more button. Our options there are to add the playlist to the queue, delete the playlist, or rename the playlist. When we click on delete playlist, we expect the following changes:

1) The playlist disappears from the playlist area.
2) We are redirected to the search area since there is no playlist to display.
3) The playlist folder inside Playlists should not exist.

To test the renaming of a playlist we need to first click on any of the existing playlists. Then, we click the rename button where we expect to see a window asking for a new name for the playlist. If we enter a name that already exists, an error window appears. If we enter a name that does not exist and is valid, we expect to see the following changes:

1) The name of the playlist in the main window is changed to the new name.
2) The songs of the playlist remain the same after renaming the playlist.
3) The playlist folder inside Playlist must have the new name.

5.3. Testing the Performance of Streaming Audio

When it comes to the performance of streaming audio, the client and server applications are expected to achieve the following:

1) Handle slow networks
2) Perform Dynamic Streaming

When the network is not fast enough to supply data for the WaveOutPlayer, the applications should not crash. The most affected application concerning slow networks is the client application because it is simultaneously playing sound. To test this, we need to simulate a slow network. The easiest way that I could think of to achieve a slow network is to slow down the loop that sends data from the server. I used Thread.Sleep() to repeatedly make the streaming thread wait for a few milliseconds. This ultimately affected how much data the client was receiving. The results were the following:

1) The server application was unaffected, and it merely kept sending data at a slower rate.
2) The client application managed to handle the slow connection by not producing any sound until the required data was available. When it had enough data to play it would automatically start playing sound.

When it comes to testing dynamic streaming, the situation is more complex. First, we need to have the client and server application running on separate computers. This is crucial because if the applications are running on the same computer the application will not initiate dynamic streaming since the data is transmitted so fast that it receives it before we can move the progress of the audio. Once we have both applications running on separate computers, we can do the following tests:

1) Change the progress of audio to 50%. We expect the audio to immediately start playing at that position. Thereafter, we leave the application running to check whether the applications will transmit the missing data.
2) Change the progress of audio to 70%, and shortly after to 30%. We expect the audio to start playing immediately. At the end the applications must fill the gaps in the stream.
3) Change the progress to any position after the current position. The audio is expected to start immediately. Afterwards, we wait until the entire stream is full. Finally, we change the index while the stream is complete. We do not expect dynamic streaming to be started since we have all the data.

The applications passed all these tests. Dynamic streaming appears to perform very well.

6. Results and Conclusions

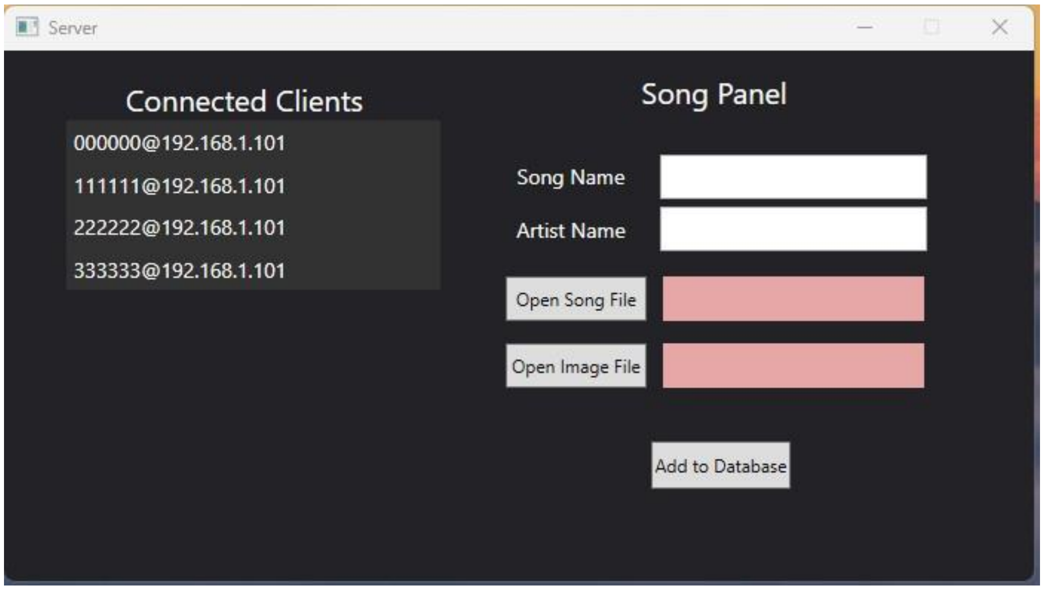
The development of a working audio streaming service has been a challenge. I am satisfied with the outcome of this project since I managed to implement all the features that I was hoping for and more. The main goal was to create a working service that can stream HiFi audio without any delays. The final applications seem to do this very well. I would say that the final applications are at a state where one can start using them as an alternative audio streaming service. My audio streaming service provides most if not all the functionalities that most users need when it comes to playing audio over the network – they can even have their own local playlists that can be shuffled if they decide to do so. The applications perform well even under slow networks.

The most challenging part for me was the development of the audio player. I had to do extensive research on what audio files provide the best audio qualities and on how to play such files. I did not expect to not find any existing audio libraries in .Net that provide low-level manipulation of sound. I faced numerous problems with wave/ out and how it interacts with .Net. Implementing a working audio player in .Net that utilizes the wave/ out multimedia functions took a longer time than expected. My initial plan was to focus more on adding features to the applications in general instead of focusing on the low-level manipulation of audio. However, the struggles that I faced, in the beginning, taught me many concepts about audio programming that I would not know otherwise.

Additionally, I came to realize that utilizing multiple threads in this project was more difficult than I expected. The many parallel tasks that the applications perform combined with the latency of networking, made the development of the audio streaming service quite complex. The synchronization of parallel tasks is a very interesting concept that I hope to learn more about in the future.

During the final step of development, it got much easier especially when I was building the graphical user interfaces for both applications. Using WPF as our graphical subsystem made the creation of user interfaces very easy.

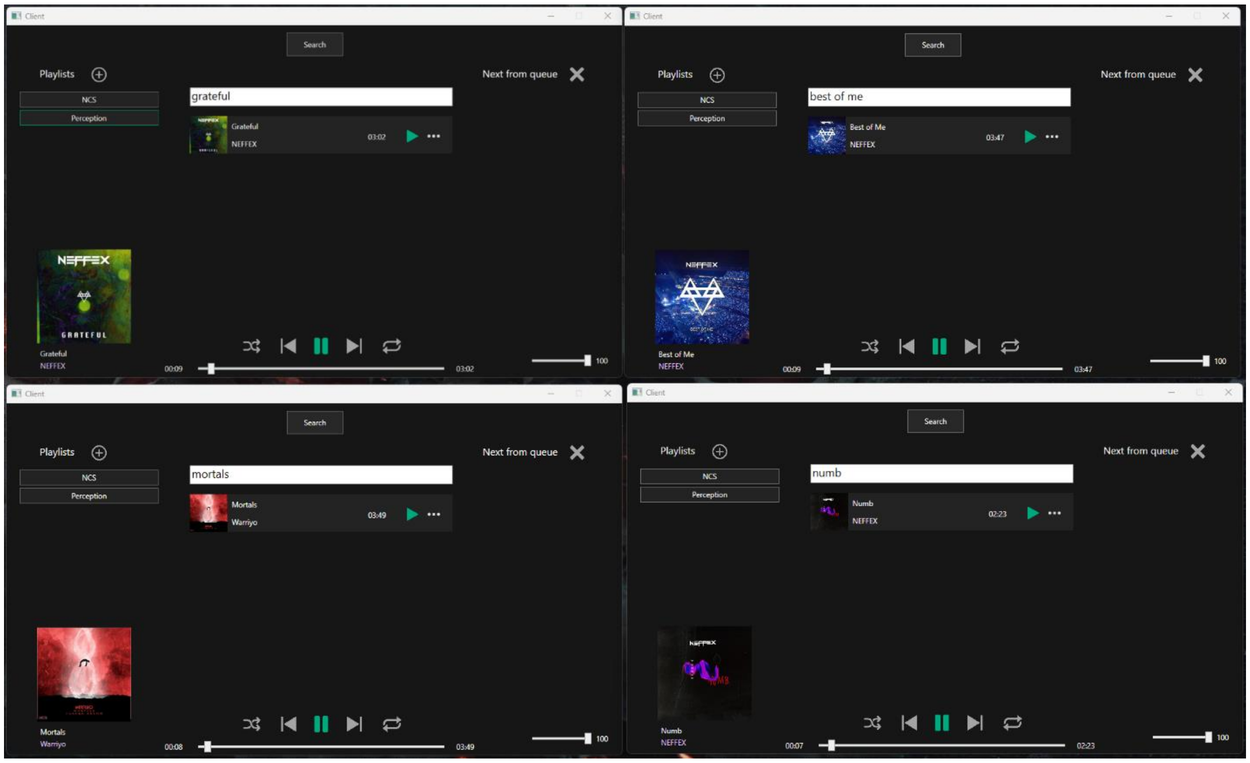
Below you can see screenshots of the audio streaming service working. There are four client applications running in one Windows machine, and a server application running in a separate machine. All four clients are streaming audio simultaneously.



000000@192.168.1.101
111111@192.168.1.101
222222@192.168.1.101
333333@192.168.1.101

Server Application Connected to Four Different Clients

On the left you can see the IDs and the IP addresses for each of the connected clients. On the right you can see the song panel where the server user can add songs in the database file and in the storage of the server.



Four Clients Streaming Songs Simultaneously

These four clients were started virtually at the same time and there was not enough time for the server to send the entire wave file to any of them. The server is simultaneously supporting four client applications that are streaming different songs at slightly different positions. Bottom-center you can see the audio player controls. On the top-left you can see the playlist area that currently contains NCS and Perception. Bottom-left you can see the song name, artist name, and image for the current playing song. Top-center you can see the search bar, and below it you can see the songs that were sent by the server after searching. On the top-right you can see the queue which is currently empty. Finally, on the bottom-right you can see the volume control which is set at 100%.

7. References

“About SQLite.” *About SQLite*, 6 Oct. 2021, sqlite.org/about.html.

“Anatomy of the Client/Server Model.” *Anatomy of the Client/Server Model*, docs.oracle.com/cd/E13203_01/tuxedo/tux80/atmi/intbas3.htm. Accessed 1 Dec. 2022.

Eddy, Wesley M. “RFC 9293: Transmission Control Protocol (TCP).” *RFC 9293: Transmission Control Protocol (TCP)*, 1 Aug. 2022, www.rfc-editor.org/rfc/rfc9293.html.

“Marshal Class (System.Runtime.InteropServices).” *Marshal Class (System.Runtime.InteropServices)* | *Microsoft Learn*, learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.marshal?view=net-7.0. Accessed 1 Dec. 2022.

“System.Net.Sockets Namespace.” *System.Net.Sockets Namespace* | *Microsoft Learn*, learn.microsoft.com/en-us/dotnet/api/system.net.sockets?view=net-7.0. Accessed 1 Dec. 2022.

“Using Threads and Threading.” *Using Threads and Threading* | *Microsoft Learn*, 4 Oct. 2022, learn.microsoft.com/en-us/dotnet/standard/threading/using-threads-and-threading.

“Wave/Out - Win32 Apps.” *Wave/Out - Win32 Apps* | *Microsoft Learn*, 7 Jan. 2021, learn.microsoft.com/en-us/windows/win32/tapi/wave-out.

“Wave and DirectSound Components - Windows Drivers.” *Wave and DirectSound Components - Windows Drivers* | *Microsoft Learn*, 15 Dec. 2021, learn.microsoft.com/en-us/windows-hardware/drivers/audio/wave-and-directsound-components.

“Wave File Specifications.” *Wave File Specifications*, www.mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html. Accessed 1 Dec. 2022.

“Windows Presentation Foundation - WPF .NET Framework.” *Windows Presentation Foundation - WPF .NET Framework* | *Microsoft Learn*, 4 Sept. 2020, learn.microsoft.com/en-us/dotnet/desktop/wpf.

.“NET (and .NET Core) - Introduction and Overview.” *.NET (and .NET Core) - Introduction and Overview* | *Microsoft Learn*, 29 Nov. 2022, learn.microsoft.com/en-us/dotnet/core/introduction.