Implementing a Highly Synchronized Multi-party Distributed Music Playing System

Eric Gong, Ryan Jiang, Evan Jiang, Charlie Chen

1 Project Overview

The rise of numerous synchronized and collaborative listening services, such as Spotify's Group Session and social audio apps like Clubhouse, reflects a growing preference by music listeners for real-time interactive audio. However, precise synchronization in distributed systems remains technically challenging. In particular, the human ear is highly sensitive to timing differences, with the average gap detection threshold being approximately 4.19 milliseconds [1]. As such, to ensure the success of a synchronous music player, it is of critical importance to develop frameworks by which clients can derive a consistent notion of time, and furthermore, to utilize this notion of time to execute synchronized actions across multiple clients. The question of how one chooses to maintain a consistent state of truth regarding what songs are being played, and when they are being played, is also of critical importance.

Utilizing a custom implementation of the Network Time Protocol, and a centralized server for relaying and propagating client requests, we construct a simple distributed system which provides a rudimentary solution to these challenges.

2 Design and Implementation

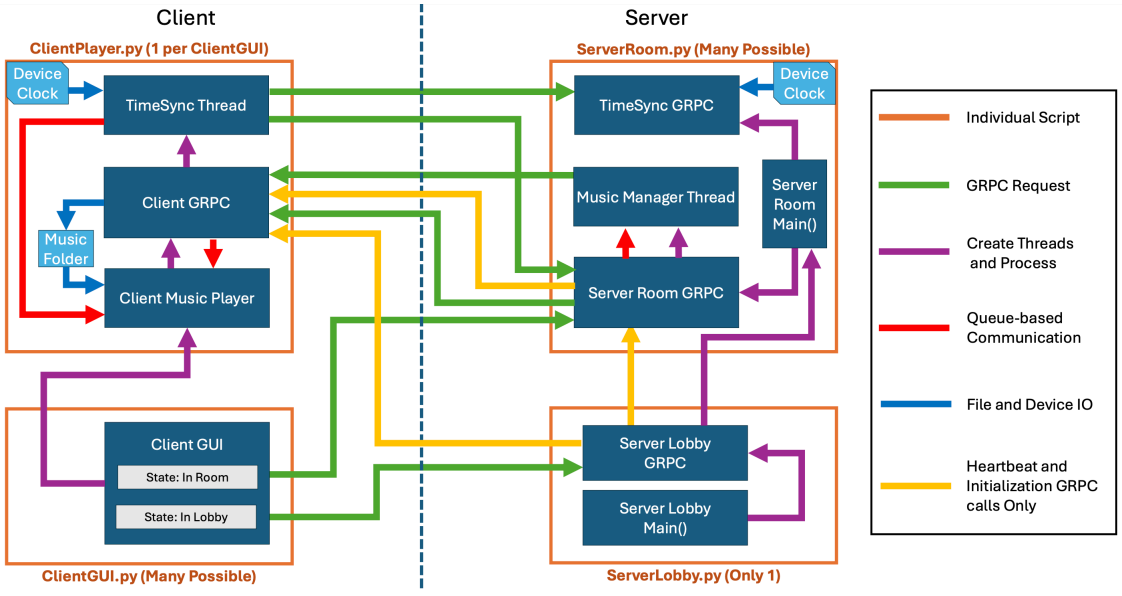

Figure 1: Schematics illustrating the various processes, threads, and communication protocols utilized in the implementation of the distributed system

2.1 High Level Client-Server Architecture

The architecture of this project is divided between a **Client**, which allows the user access to the distributed system, and a **Server** side, which accepts request from clients, and relays commands back to all clients as necessary.

The server consists of a single Server Lobby, implemented in `ServerLobby.py`, which accepts requests from Clients to start up Server Rooms as separate processes, where each Server Room is implemented in `ServerRoom.py`. We choose to start up new rooms as new processes given that it lays down a groundwork for supporting server-side replication, which is one potential future extension. One could imagine that Server Rooms could be started up on a different device than the Server Lobby, which would require independent processes. However, for the purposes of this exercise, given time constraints and constraints in fully testing such a system with a limited number of devices, we choose not to pursue replication on the server.

The Server Lobby is in charge of maintaining a list of active Clients, as well as active Server Room processes. Each Server Room is responsible for maintaining an active connection all Clients, as well as maintaining the consistent state of truth for that room, including the queue of songs to be played, the current song, whether the song is currently playing or paused, as well the current timestamp position of the playback.

The Client is split between a Client GUI and a Client Music Player. The Client GUI allows the user to send requests to the server, modifying the state of truth contained on the server side: requesting to start new rooms; requesting new songs be added to the queue; requesting that a certain song start playing, stop playing, or be skipped, etc. The Client Music Player is primarily reactive, responding to declarations in state changes by the Server Room, particularly as it pertains to changes in the music state. The Client Music Player does, however, actively communicate with the Server Room for the sake of executing our custom implementation of the Network Time Protocol, which is elaborated on below. There should exist only one Client Music Player per Client GUI, and the two should always be on the same device, as such, we run the Client Music Player in a thread that is started by the Client GUI. We choose to modularize our code into the Client GUI and Client Music Player for simplicity of debugging and testing.

As seen in Figure 1, all communication between the Client and the Server occur through gRPC calls. Communication between different thread within the same process, either in the Server Rooms, or on the Client side, occur through the usage of Queues, or File-IO, implemented such that one thread solely writes data, and the other solely reads. This design choice, alongside careful and meticulous planning, ensures that outside of the creation of thread and processes, the data flow and initiation of requests (which is denoted by the direction of the arrows in Figure 1), forms a directed acyclic graph. This provides a provable guarantee that it will not be possible for any action within the system to result in deadlock (which typically occurs due to two gRPC servicers attempting to issue requests to one another while waiting for a response from the other at the same time), or an infinitely propagating loop of commands. As such, this ultimately simplifies the process of testing and debugging.

2.2 Server Room Architecture

The Server Room is constructed with three main components: a TimeSync gRPC servicer, a Server Room gRPC servicer, and a Music Player Manager thread. It is necessary to have both threads and gRPC servicers given that a thread is necessary for the active roles the Server Room plays (sending out commands to the Client Music Player), whereas the servicers are responsible for the reactive role, responding to incoming requests from clients.

The Music Player Manager thread maintains the state of truth for songs that are to be played, stored within a Song Queue, as well as the state of the current song: whether or not the song is playing, the timestamp position to be played at any given time, and a bytes object of the song itself. The Music Player Manager is responsible for sending out gRPC requests to all Clients Music Players, issuing instructions to start or stop songs. Notably, while skipping songs is also a functionality offered to the Client, it does not require a separate gRPC implementation; the start song command specifies the song to play and the position to play at, so skipping a song is equivalent to issuing a start request which specifies the next song with a position of 0.

The Server Room gRPC servicer is responsible for accepting requests to change state in the Music Player Manager. The Server Room gRPC servicer is also responsible for responding to the Client GUI on State requests, providing information on the song queue and active users. Finally the Server Room is also responsible for aiding the process of a Client that joins the room, which involves connecting to the Client Music Player, providing the address of the TimeSync gRPC servicer, and also updating Client Music Player state on the current list of songs. To ensure that all requests are addressed in a timely manner, we allow the servicer to be multi-threaded. However, this introduces the additional complexity of concurrency in requests for updating state in the Music Player Manager. To mitigate this, we utilize a lock, such that only one action among the list of Start, Stop, Skip and Add song can be serviced at any time, but an arbitrary number of Join Room requests and State requests can be answered at any time, subject only to the number of threads instantiated. Any gRPC request that fails to obtain the lock returns immediately with a failure notification response to ensure no delays and blocking. In addition, we also implement a slight delay on the release for the lock, such that commands cannot update Music Player Manager state too often. The lock is passed to a thread whose sole purpose is to release the lock. The thread is necessary so that the gRPC request can return a response as soon as possible, while the lock is still held. The simplicity of the thread ensures that the lock will always be released. Furthermore, given that the thread is only ever called by a request that holds the lock, we are guranteed that there will ever be one such thread active at any given time.

The TimeSync gRPC servicer is highly simplistic. Its sole purpose is to enable the custom implementation of the Network Time Protocol for time synchronization by returning the current system time to any request it receives. We choose to multi-thread this servicer, and to keep it as a separate servicer from the Server Room gRPC servicer, to further ensure that there is no backlog in either the TimeSync requests or other requests to updated the Music Manager state, both of which should occur with as little backlog and delay as possible.

2.3 Client Music Player Architecture

The Client Music Player is composed of a TimeSync thread, a Client gRPC servicer, and a Client Music Player thread.

The Client gRPC receives commands from the Server Room gRPC servicer and Music Manager thread, with commands to Add, Start, and Stop songs. The gRPC responds to these requests with empty responses, and the Server does not check the responses; for the sake of simplicity and efficiency during execution, we offer at most once delivery of commands to the Client. If the Client misses a command to start or stop, that will not be an issue, as the Client will simply be re-synchronized with other clients upon the next Start or Stop request. If the Client misses a request to Add a song, the song will not play for the Client. In this case, the client can either skip the song (which will skip the song for all other Clients), or re-join the room. We choose not to allow the Client gRPC servicer to issue song bytestring re-requests given that it is slow to send large files relative to the speed at which the client could re-request, and in addition, doing so would cause the data and request-flow graph (Figure 1) to become cyclic, removing the provable guarantee of no deadlocks or self-sustaining loops of activity. The Client gRPC also receives a request from the Server Room whenever the Client GUI end chooses to join a room. The request contains the address of the Server TimeSync gRPC servicer, and the Client gRPC will terminate any old TimeSync threads, and start a new TimeSync thread that will connect to the new room. (The purpose of the TimeSync thread will be covered later on.)

The Client gRPC also issues commands to the Client Music Player thread via a Queue. The Client Music Player is the thread which ultimately plays the actual music for the Client to hear. To ensure the lowest latency responses in music controls, we utilize the python wrapper around the VLC media player. The Server Room will specify an action, when the action should take place, and additional information about the action, which will be received by the Client servicer, and ultimately acted on by the Client gRPC. In particular, as will be elaborated on later on, our custom Network Time Protocol implementation ensures that based on the action start time provided by the Server Room, it is possible to calculate the corresponding time relative to the Client's system clock, which ensures that all Client Music Players execute the action at the same time in reality, despite any drift or deviation inherent in each Client's clock, relative to the server.

2.4 Custom Network Time Protocol Implementation

The greatest technical challenge in this distributed system is to ensure that there is a consistent notion of time, such that Clients are able to execute actions that will occur at the same time in reality.

To accomplish this, we utilize a custom implementation of the Network Time Protocol. In particular, there are two main estimands we wish to quantify: the time required for a one-way transmission of a gRPC request, which we denote as the **Delay**, and the difference in value between the Client's system clock and the Server's system clock at any point in time, which we denote as the **Offset**.

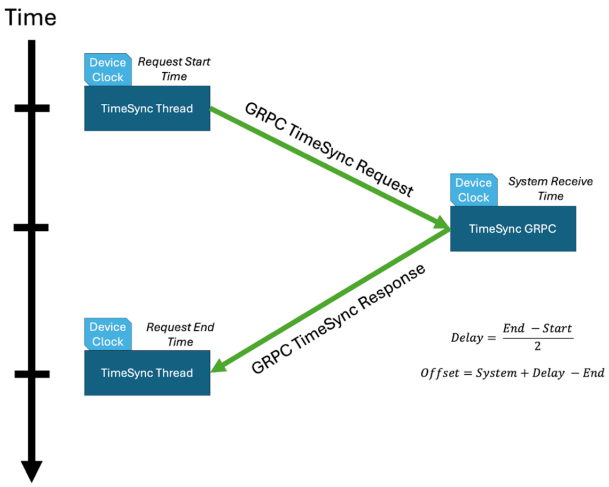

Figure 2: Custom Network Time Protocol implementation

Our custom implementation is relatively simple, and involves a request initiated by the Client TimeSync thread, and responded to by the Server Room's TimeSync gRPC servicer. In particular, the Client TimeSync thread begins a time synchronization exchange by logging the current Client system time, and sending out a TimeSync gRPC request to the Server Room TimeSync servicer. The servicer, upon receiving any TimeSync request, immediately responds with the current Server system time. Finally, upon receipt of the response, the Client TimeSync thread logs the receive time. The Client now has three measurements: the start time of the request, the end time of the request—both of which are measured with respects to the Client system time—and the time at which the Server received the request, with respect to the Server system time. The two measurements made by the client allow the client to calculate the round-trip network latency, which when divided by two, roughly approximates the delay between the send and receipt of a gRPC request. The measurement done by the Server allows the Client to approximate the relative difference in system clock time between the Client and the Server. We note that this calculation makes the assumption that the Server makes its time measurement at exactly the halfway point between the start and end times logged by the Client. In practice, we find this is not the case, and that in fact, this assumption introduces a roughly 0.5 millisecond bias when the client and server are run on the same device with the same system clock. That is, while we would expect an offset of 0 to be calculated, the incorrect assumption introduces a bias in all clients. It is important to note however, that this bias is consistent across all clients, and so it will not affect our ability to ensure that all Clients have the same notion of time relative to the Server system clock.

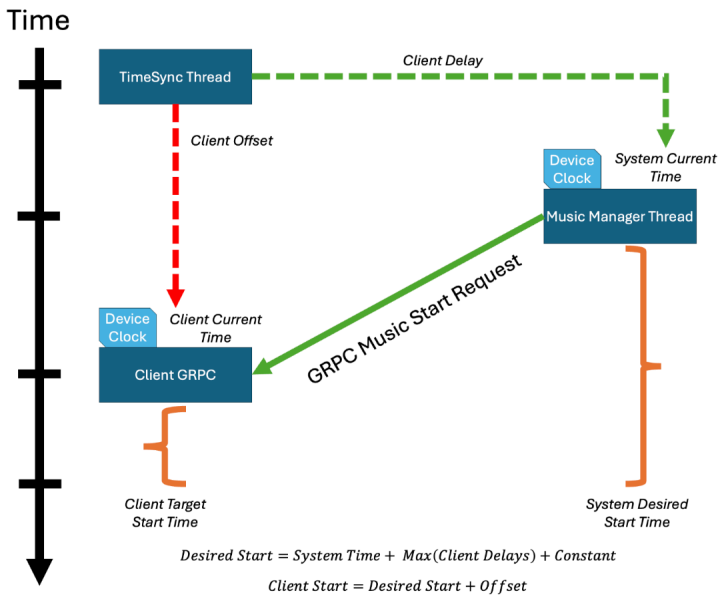

Figure 3: Command delivery diagram.

After the delay and offset are measured by the client's TimeSync thread, the delay is conveyed back to the Server Room's Music Manager thread through the Server Room gRPC servicer, and the offset is conveyed to the Client Music Player. The Server Room will then possess the network communication delays across all clients, and has the ability to calculate a start time for commands projected to take place in the future, such that if the command were sent out now, all clients would receive the gRPC request to invoke the action, and have the time to do any pre-processing, before the start time arrives. We can see in Figure 3 that the server uses the maximum across all collected client delays, and adds a constant to determine how far in the future the action's start should take place, before sending out the gRPC request to all the clients containing the action and the start time. The Clients receive this start time, which is relative to the Server System Clock, and use the offset that was calculated prior to determine what time this would be in the Client's system clock, ensuring that users perceive a synchronized execution of actions across all Clients.

We note that to ensure the TimeSync thread gets an accurate measurement of delay and offset, and also does not flood the Server Room Music Manager with update requests, we implement the code such that the TimeSync thread calculates the average offset across recent TimeSync exchanges—where the number of most-recent offsets to use is a constant located in `ServerConstants.py`—and the maximum delay is taken across these most recent offsets. We only allow the TimeSync thread to update the Client Music Player or the Server Room Music Manager if the offset or delay differ from the previous measurement by a large enough threshold, which also based on a constant defined in `ServerConstants.py`.

2.5 Audio Chunking

Initially, audio chunking was considered to reduce playback delays, but comprehensive tests demonstrated optimal performance by uploading entire audio files at once, negating the benefits of chunking. The idea was that we pre-load chunks of X seconds at a time. There are some benefits and drawbacks of this:

- Pros: Chunking reduces the effective upload time of a song, simplifies the PlaySong functionality of the server (since it just includes which chunk we're supposed to play), and is used in industry standard (see YouTube buffering or Zoom Video/Audio catchup)
- Cons: If the upload time of an individual chunk exceeds the chunk length, there will be audible delays in music playing. In addition, we need exact precision of where we are to avoid "in-between" chunk errors. Lastly, pausing and playing might reset a chunk, causing music to be replayed.

Furthermore, even if we did choose to chunk, we had to decide whether we should have fixed chunk lengths (e.g. always be 5 seconds at a time), or if we should have dynamically sized chunks (e.g. always $\frac{1}{50}$th of a song's length).

Based on the figures below though, there appears to be a fixed cost in uploading chunks, indicating that the overhead does not counteract the optimizations. In addition, we found the optimal chunk size to be the same, regardless of all song lengths we tested (visually separated by the graph), indicating that fixed sizes were the way to go, rather than dynamic sizes.

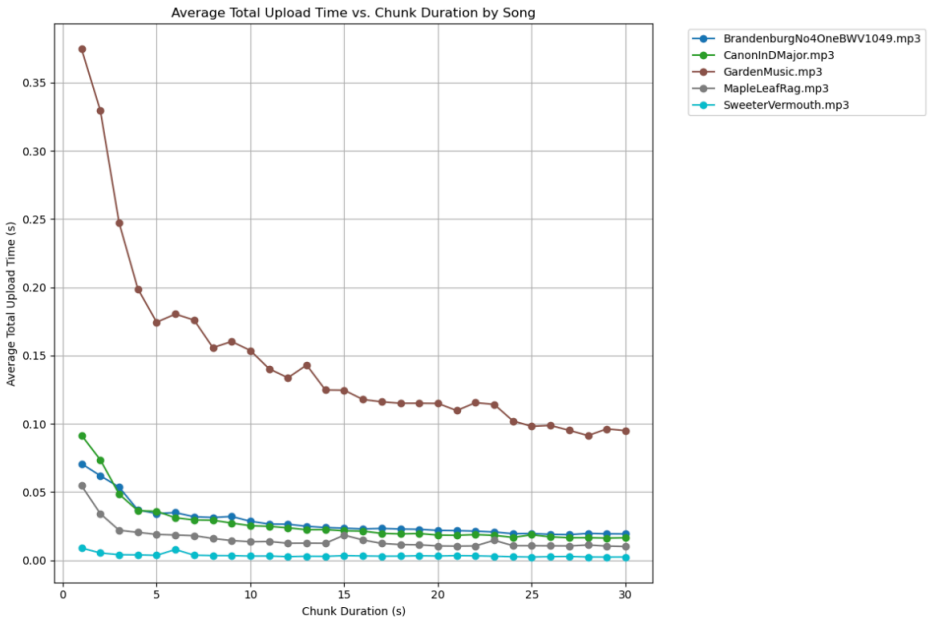

Figure 4: Average total upload time on chunk duration.

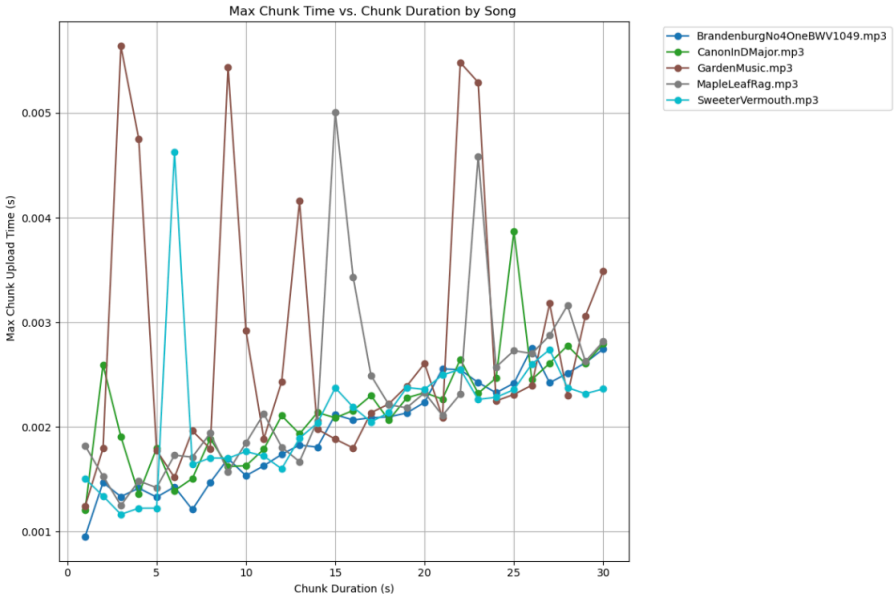

Figure 5: Max chunk time on chunk duration.

2.6 GUI and Full-Stack Design

The frontend interface was built with a focus on simplicity and robustness. The backend was implemented using gRPC to manage communication efficiently. Figures 6 and 7 show the components on the frontend and how they connect.

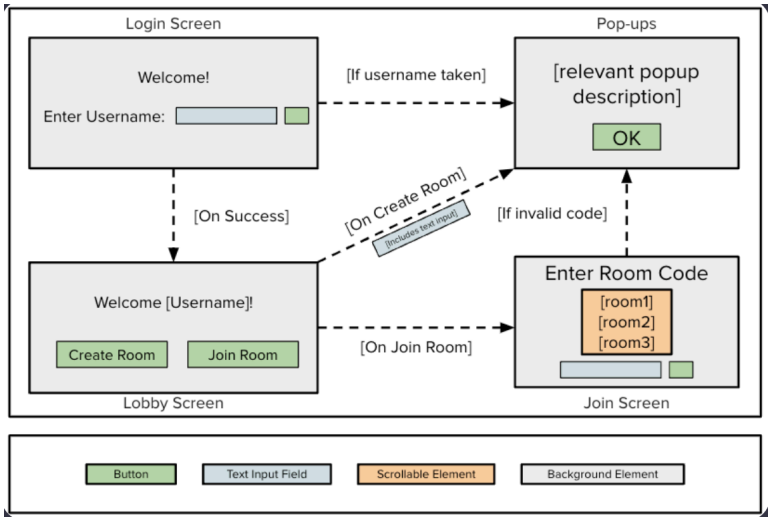

Figure 6: Logical flowchart for User Interfaces related to Login, creating Rooms and joining Rooms

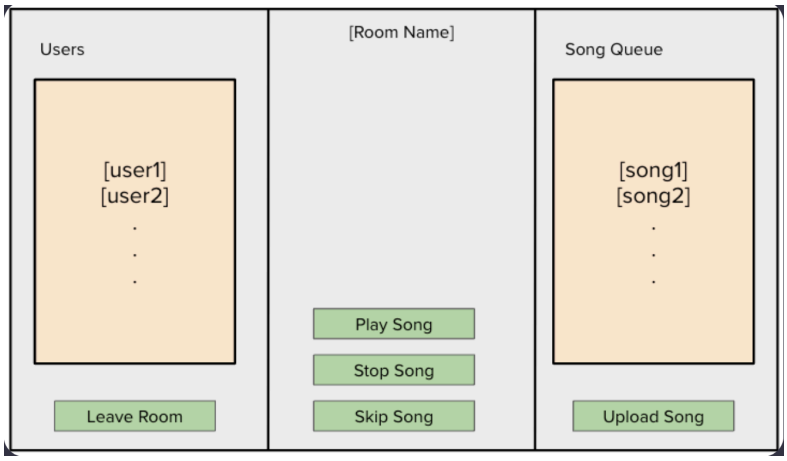

Figure 7: Interface with a suite of functionalities for the user, enabling one to add, start, stop and skip songs

3 Testing

We developed a comprehensive test suite covering both unit and integration tests for the gRPC-based music player service, focusing on the ServerLobby and ServerRoom components.

Test Structure

The tests are organized into five main classes:

- **TestServerLobbyServicer**: Unit tests verifying lobby functionalities such as user join/leave, room creation, and duplicate username handling.
- **TestServerRoomTimeServicer**: Unit tests ensuring accurate server-client time synchronization responses.
- **TestServerRoomMusicServicer**: Unit tests verifying music room operations, including adding songs, starting/stopping playback, and handling user activity.
- **TestServerIntegration**: Integration tests checking interaction between the lobby and room services (room creation and joining).
- **TestMusicPlayerIntegration**: Integration tests validating the full music playback flow, including song queuing and playback control.

3.1 Testing Techniques

We heavily used the `unittest.mock` library to mock gRPC channels, file operations, and thread behavior, enabling isolated and reliable testing without relying on external systems. Patch decorators and context managers provided fine-grained control over mocked behaviors. Special attention was given to threading safety, avoiding race conditions during concurrent operations.

3.2 Addressing Challenges

Key challenges included simulating gRPC communication without network dependency, handling concurrency safely through mock threading, testing file upload behavior without actual MP3 files, and verifying error handling under simulated network failures. Our testing strategy ensures that both individual components and system-wide interactions behave correctly under realistic and edge-case conditions.

4 Timeline and Milestones

April 8th

- Selected libraries (brainstormed): Tkinter, gRPC, cachetools, Pydub, pygame, Asyncio.
- Defined initial architecture and basic design.
- Considered initial chunking strategy.
- Established core backend and frontend interactions.

April 14, 2025

- Implemented initial server functionalities (jam room creation, basic command handling).
- Set up time synchronization protocol similar to NTP.
- Documented detailed specifications for server commands.

April 15, 2025

- Began user interface development.
- Defined client-side commands: AddSong, SkipSong, StartSong, StopSong.
- Decided against chunked audio streaming due to performance testing results.
- Clarified separation between GUI and backend components.

April 16, 2025

- Expanded GUI-backend separation further.
- Implemented client-side polling for dynamic updates (room participation, song queue management).

April 21, 2025

- Integrated additional libraries: grpcio-tools, Python-vlc, Mutagen, Numpy, PyQT5. Dropped old GUI ideas.
- Identified and resolved synchronization issues.
- Ensured correct audio offsets for users joining active playback sessions.

April 22, 2025

- Completed comprehensive test suite (backend unit tests, lobby-room integration, client functionality).
- Conducted detailed demo planning.
- Outlined future enhancements: GUI updates, authentication mechanisms, caching strategies, fault-tolerant persistent storage.

5 Known Limitations

System performance decreases as the number of users in a room increases. In particular, the communication and synchronization overhead with each additional client can introduce noticeable latency and minor desynchronization during playback. At present, the system exclusively supports MP3 files, limiting user flexibility for higher-quality or alternative audio formats. Finally, when a new user joins an ongoing session, that user will not be able to hear music immediately. Instead, they will hear music only upon the next start (either when transitioning to the next song, or more reasonably, after the current song has been paused, and started again). We choose such an implementation to ensure user experience while maintaining simplicity. If we had chosen to start the new user's music immediately, that would result in unfavorable situations that become technically challenging to mitigate. If we chose only to modify the state of the new user, it may start off-sync with the other users. On the other hand, if we choose to synchronize a re-start to ensure all users are at the same point, that could potentially cause stalls or skips in the audio experience of current users every time a new user joins. Both cases are unfavorable, as such, we only start playing music for new users when a new song plays, or when the current song is re-started after being stopped.

6 Future Work

Several enhancements could be possible to improve system robustness and user experience. Future work could consider expanding the GUI to display detailed song metadata such as title, artist, album, and playback progress. Persistent storage will be implemented to allow jam rooms and queues to survive server restarts. Command replication across backup servers will be added to

improve fault tolerance and prevent data loss. Caching frequently played tracks locally will reduce latency and improve playback smoothness. Finally, a user authentication and login system will be introduced to prevent username conflicts and better manage user sessions. Together, these upgrades will enhance scalability, resilience, and overall usability of the music player.

References

[1] Alessandra Giannela Samelli and Eliane Schochat. "The gaps-in-noise test: gap detection thresholds in normal-hearing young adults". In: *Int. J. Audiol.* 47.5 (May 2008), pp. 238–245. DOI: 10.1080/14992020801908244.