CS 2620 Project Proposal: Spotify Jam

Charlie Chen, Eric Gong, Ryan Jiang, and Evan Jiang

April 4, 2025

Motivation

Music streaming services like Spotify allow users not only to listen individually but also to listen together in real-time through features like *Spotify Jam*. These shared listening sessions involve synchronized playback, shared queues, and multi-user control, which introduce interesting distributed systems challenges. This project aims to build a simplified version of such a system to explore how concepts like time, server-client communication, and propagation can support a user friendly musical streaming environment.

Project Goal

Build a small-scale music streaming backend that:

- Supports multiple clients streaming songs individually.
- Allows users to create or join "Jam Sessions" where everyone listens to the same music at the same time.
- Lets all users in a session contribute to a shared song queue.

Core Features

1. Individual Streaming

Clients can connect and stream songs independently, downloading them chunk by chunk. Playback is smooth due to pre-buffering and caching on the server.

2. Jam Sessions (Shared Listening)

Users can create or join a "Jam" (shared listening session). Everyone in the same Jam hears the same song at the same time (synchronized playback). Users can add songs to a shared queue, and everyone hears the next song when the current one ends.

3. Chunked Streaming

Songs are divided into small chunks (e.g., 5 seconds). Clients prefetch chunks to avoid playback gaps. The server manages which chunk each user or Jam session should be playing at a given time.

Simplified System Design

```
Clients → Backend Servers → Song Storage
                  ↓
(In-memory Cache) → Jam Session Coordinator
```

Each Jam session maintains shared state: current song, playback time, and queue. The coordinator handles syncing playback and queue changes across all clients in a Jam.

Technologies

- Language: Python
- Communication: gRPC or HTTP (supporting streaming and sync updates)
- Storage: Local folder with audio files (MP3/OGG)
- Cache: In-memory dictionary with LRU eviction policy
- Jam Sync: Polling or lightweight push updates for playback synchronization

Design Choices

There are many design choices to consider. We will list a few preliminary ones here:

- Buffering on commands: To synchronize commands from individual clients affecting the entire jam, we choose to have some buffer that all other clients will recognize. This means we're utilizing "at most once" style messaging and consider rogue machines missing the command a failure. For example, if one machine pauses, all other machines should expect to pause in X time, trading off immediate results for coordinated ones.
- Catchup and fallback protocol: If a client does fall in and out of the jam, they may not be in sync with the rest. We choose to implement jumps rather than speedups/slowdowns (unlike, say, Zoom meetings which freeze and then fast forward or slo-mo audio). This trades off a sharper cutoff in exchange for avoiding the "record scratching" sounds associated with continuous re-integration.
- Server-client system: We will have our main server be responsible for holding the true state of the jam (e.g. play time, queue, etc...). This opens the door for replication of servers if we want fault tolerance, and we will propogate all commands from the server rather than doing anything P2P. This trades off some speed for coordination efforts.