ClouDJ: A Music Sharing Application

Amanda Hittson, Rebecca Lam, Josh Slauson
University of Wisconsin-Madison

ABSTRACT

ClouDJ enables music sharing. Listening to music is largely a social activity that is best enjoyed with others. However, there is currently no easy way to listen to music together online. We plan to make listening and sharing music easier with our system ClouDJ. ClouDJ will allow users to play their own music while sharing it with other users in real-time.

1. INTRODUCTION

For our project we designed and implement ClouDJ, a music sharing application that uses a centralized server system. The ClouDJ application allows multiple clients to simultaneously listen to music together in shared sessions. To this end, we designed a system with the following goals:

- Performance: users that are participating in a session together should be able to listen to music at a similar rate.
- Availability: users must be able to continuously listen to music in a session assuming the user hosting the session is available.
- Scalability: adding clients should not affect system performance.

2. DESIGN

To create ClouDJ, we implemented the design described in subsections 2.2 and 2.1 by integrating Google App Engine. Figure 1 depicts the overall system.

Figure 1: The system architecture.

2.1 Roles

There are three major roles in our system: the master handler, session handler, and client. Clients own and store music, can share their music with others and can listen to music shared by others. The master handler provides access control, starts sessions for users who want to host, and acts as a liaison between the clients and session servers. The session handler is in charge of handling synchronization messages between clients and serving content.

2.2 Front End

A session is an abstraction in which multiple users may listen to one song that is hosted by a single user. Our front end application allows a user to either create a session and become a host, or join an existing session and become a listener. When acting as a host, a user may add or remove songs from the session playlist, play or pause the current song, and skip to the next song. When acting as a listener, the user has no control over what song is being played. Users can see all sessions that they are able to join. Only members of a user's access control list (ACL) may see or join any session in which that user is a host.

2.3 Back End

The backend infrastructure is more complicated than the frontend. The master server keeps track of users currently online, user ACLs, and user membership lists (ACLs it is a member of). It also is responsible for maintaining a session table, a table that maps host users to sessions (this is a one-to-one mapping). When a client logs on, the master server informs each session associated to a host on the client's membership list that this client is a potential listener.

The session handler is the workhorse of the system. It services requests for sessions it is in charge of by routing data from the host client to relevant clients (listeners). It maintains the list of listeners and potential listeners. A potential listener is a client who could listen to this session, but is currently not. In other words, these are the clients on the host client's ACL that are logged in but not listening to this session. Session servers also take care of session cleanup when a session ends (or fails).

Finally, the client exists on the user machine has access to its user's music and playlists and also keeps track of data such as the user's current session and the user's potential sessions (sessions this user can access). For the current session, the client keeps track of the current session key and relevant song information. For the potential sessions, the client keeps the host, session server address, and currently playing song.

Sessions are created when a client contacts the master handler about hosting a new session. The master handler then creates a new session in memory and responds to the client, which updates its current session information. Clients may join sessions by contacting the master handler, which updates the session information in memory and notifies everyone in the session. Notice that after a session is established, the client no longer communicates with the master handler and the master handles no music data.

3. IMPLEMENTATION

3.1 Client and Server Communication

Our client is written in JavaScript and uses SoundManager, a JavaScript audio player API. In contrast, our server code is written in Python 2.7. Both use the Google App Engine API.

When the client wants to contact the server, it issues an HTTP request, which is handled by the appropriate Python script handler. Handlers in Google App Engine can only serve requests within a time limit, which is why we chose to store session data within the Datastore.

In contrast, when the server wants to contact a client, is uses the Channel service. Channels are persistent connections that allow updates to propagate to clients without the use of polling. This is the mechanism with which we send updates to participants of a session. A channel is created as follows:

1. The client contacts the server
2. The server generates a channel ID and sends back a token
3. The client connects to the channel using the token

The client receives updates by listening on the channel by opening a socket and does appropriate actions based on the message received. Updates to the channel are made via HTTP requests from clients. When a client disconnects from the channel, a message containing the corresponding channel ID is sent to the server, and appropriate cleanup is performed.

3.2 Execution Flow

3.3 Session Synchronization

In order to synchronize playback among different clients, we use control messages, which are propagated via channels as described above. Only hosts may issue commands such as play, pause, and next for the session. The “play” and “pause” commands are sent only when the host clicks on the pause and play buttons, and the “next” command is sent when the host switches songs or if the host clicks on the “next” button. We separate the problem of synchronization into two parts: joining an existing session and updating the current session. In both cases we utilize a timestamp, which is updated each time a play or next command is sent by the host client and is sent with the control message.

As seen in Figure 3.3 when a user joins a session, the server sends the user the elapsed time since the last stored timestamp. The user then start the song with the value returned by the server as the offset. If the song is playing when the user joins the session, it will start playing at the specified offset. If the song is paused when the user joins the session, the retrieved offset will not be the correct elapsed time; however, this is corrected when the host sends a “play” message to the server, which stores another timestamp and propagates the correct offset to all session listeners.

This timestamping synchronization is used mainly when a user joins a session. In the other case where participants update their current session upon recieving a control message, the offset can usually be inferred based on the local state. For instance, if the user receives a “next” message, then they start the next song with an offset of 0. If a user receives a “pause” message, they stop the current song and keep the elapsed time as the offset. If the user receives a “play” message they start the current song at the local offset, which was set when the “pause” message was received. The exception to this is if the current paused song is at the beginning, indicating that this user has joined the session while the song was paused or that this user is ahead of the host and is waiting for a control message from the host. In this case the user adjusts the current song offset with what they recieve from the server. Conversely, if the listener lags behind the host, then they will receive a “next” message before the end of the song and start the next song. Figure 3.3 illustrates the timeline in the cases where a listener is behind the host and where a listener is ahead of the host.

Figure 2: The synchronization timeline for a) joining a session and b) updating a session

3.4 Design Considerations

Performance: buffered songs on client Scalability: incremental messages: listeners only on song transitions or join of session Availability: (failure tolerance) - Session info stored on Datastore - As long as google app engine up, our app is up

4. EVALUATION

We will evaluate our system on the metrics of performance, availability and scalibilty. To measure performance, we will design benchmarks to measure the data rate of a listener versus the data rate of the host. To evaluate availability, we will examine how our system behaves with various failure modes by artificially taking down each role (client, master server, session server). For scalability, we will measure system performance as we add clients to the system. We can achieve this by launching our application on an increasing number of client machines.

5. FUTURE WORK

<ADD FUTURE WORK INFORMATION HERE>

6. CONCLUSION

We will be developing a music sharing application called ClouDJ. With ClouDJ, users will be able to host music sessions that other users can listen to in real-time. It will consist of many client users as well as some centralized servers to handle session and user state. We will make use of Google App Engine for the implementation of the system.

7. REFERENCES