ClouDJ: A Music Sharing Application

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

ABSTRACT

While there are plenty of distributed music streaming applications, there is a surprisingly limited selection of standalone services for listening to music with others online. Thus, we present the design and implementation of ClouDJ, a distributed application that enables multiple users to listen to songs at the same time in a shared session. ClouDJ was designed with the goals of performance, availability, and scalability in mind. This is achieved through synchronization, use of Google App Engine services, and concurrency.

1. INTRODUCTION

For our project we designed and implemented 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. This is similar to Google Hangouts [2] and Spotify [6] You can see our application online [3] as well as our source code [4]. 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.

The rest of the paper is structured as follows. In Section 2 we discuss the architecture and design of the underlying components. In Section 3 we examine the implementation details and how we adapt the design to the Google App Engine framework. In Section 5 we evaluate our system based on the aforementioned performance goals, and in Section 6 and Section 7 we finish with future work and concluding remarks.

2. DESIGN

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

2.1 Architecture

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 connects users to their desired session. The session handler is in charge of handling synchronization messages between clients and serving content.

2.1.1 Front End

A session is an abstraction in which multiple users may listen simultaneously to one song that is hosted by a single user. Our front end (client) 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 songs to 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.

The client 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 session key and relevant song information. For the potential sessions, the client keeps the host, session key, and currently playing song.

2.1.2 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 adding entries to the session table, a table that keeps track of session information including host, listeners, and current song. It also maintains the potential listener list for each session. A potential listener is a client who could listen to the 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. When a client logs on, the master server retrieves each session associated to a host on the client's membership list and sends back a list of sessions it could potentially join. It also adds the client to the potential listener list for each of those sessions.

The session handler is the workhorse of the system. It services requests for all sessions by routing data from the host client to relevant clients (listeners). It is in charge of maintaining the data in the session table and propagating commands from the host to all listeners. It is responsible for handling uploads and serving song data. The session handler also takes care of session cleanup when a session

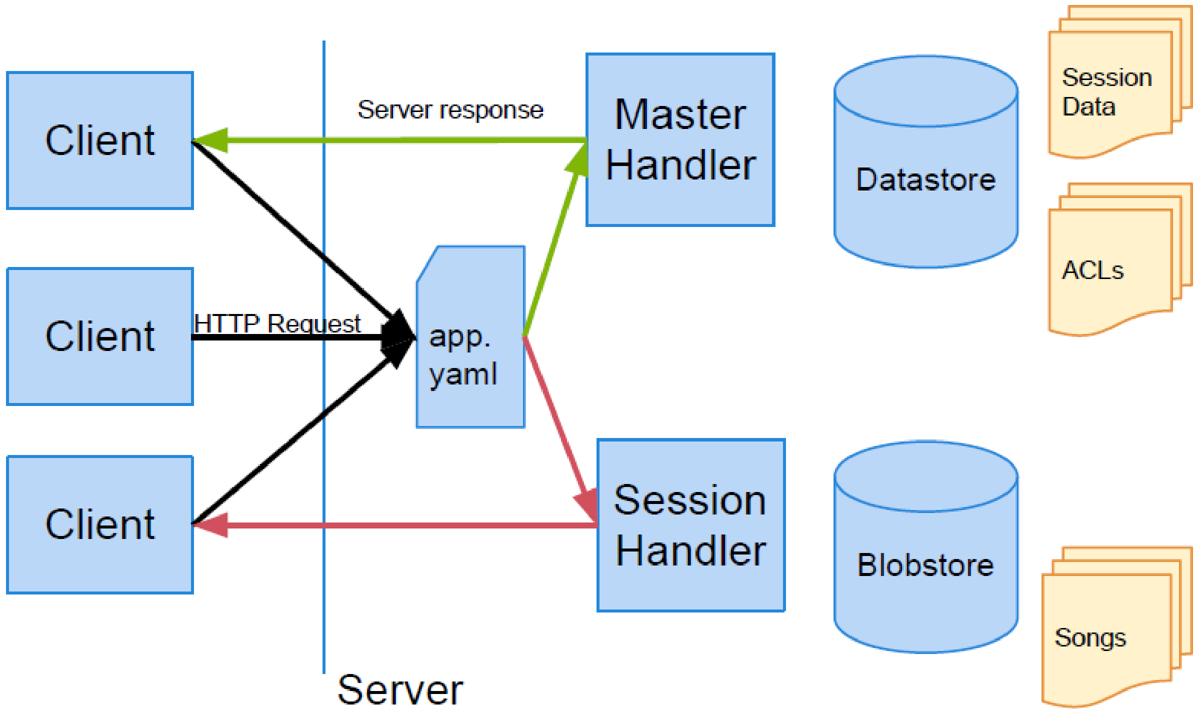


Figure 1: The system architecture.

ends (or host disconnects).

2.1.3 *Storage*

Certain information is stored on the server in the Datastore and Blobstore. Datastore holds session data and ACLs. Session data includes session meta data (session key, host, listeners), and session state (current song, song play or pause flag, session ended flag, timestamp of last play/next message). This information is used to coordinate sessions among different clients. The ACLs keeps track of the potential sessions and potential listeners for each user. Potential listeners are users who can listen to a session hosted by a given user and potential sessions are sessions the user can join. Blobstore is only used to store song data in our application. When a session ends, the songs are deleted from the blobstore.

2.2 Execution Flow

As shown in Figure 1, the general execution of the ClouDJ application is as as follows:

1. The client contacts the server
2. The server executes the handler based on the request
3. The handler runs and propagates updates to session participants
4. The client receives the server response and performs actions

In sections 2.2.1 and 2.2.2, we discuss session creation, joining and leaving sessions, and updating sessions.

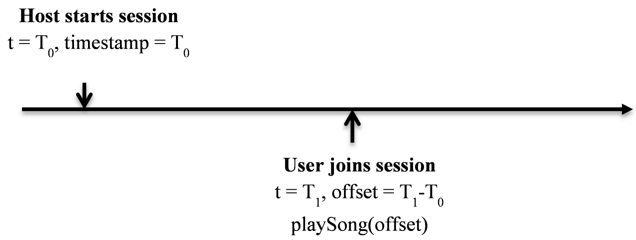
2.2.1 *Creating, Joining, and Leaving a Session*

Sessions are created when a client contacts the master handler about hosting a new session. The master handler then creates a new session in the datastore and responds to the client, which updates its current session information. Clients may join sessions by contacting the master handler with the session key of the session they wish to join. The master handler updates the session information in the datastore and notifies everyone in the session of the new listener. After a session is established, the client no longer communicates with the master handler. When a user leaves a session, the client contacts the session handler telling it to remove itself from from the listener list if it is not the host. If it is the host, the session handler notifies all listeners that the session has ended and removes the session and corresponding song data from the datastore and blobstore.

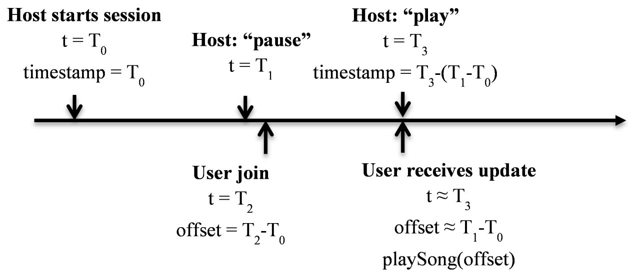
2.2.2 *Updating Sessions*

Hosts may update the sessions using several types of messages: “upload,” “play,” “pause,” and “next.” Songs are added to the session when the host client contacts the session handler with an “upload” message, indicating that it wants to add a song to the playlist. The session handler forwards the song to the blobstore and retrieves the corresponding blob key, which is used when requesting the song for playback. The session handler then updates the session playlist in the datastore and sends the blob key to all clients in the session. The clients then fetch the song from the blobstore.

Unsurprisingly, the “play,” “pause,” and “next” messages indicate when the host wants to unpause, pause, and skip the current song, respectively. The host sends these control messages to the session handler, which updates the session in the datastore and then propagates the message to all ses-



(a) Joining a session while a song is playing



(b) Joining a session while a song is paused

Figure 2: Synchronization messages when a user joins a session

sion participants. The clients then peform the appropriate action. We elaborate on playback synchronization among clients in Section 2.2.3.

2.2.3 *Session Synchronization*

In order to synchronize playback among different clients, we use control messages, which are propagated as described in Section 3.1. 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 2 when a user joins a session, the server sends the user the elapsed time since the last stored timestamp. The user then starts 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

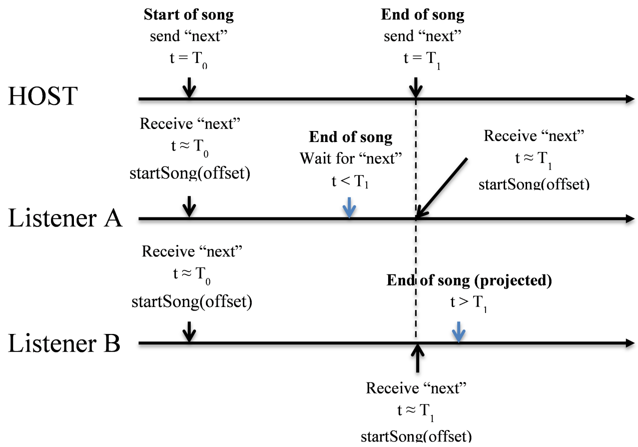


Figure 3: The synchronization timeline for a listener that's ahead of or behind the host

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 illustrates the timeline in the cases where a listener is behind the host and where a listener is ahead of the host.

3. IMPLEMENTATION

Our client is written in JavaScript and uses SoundManager, a JavaScript audio player API [7]. In contrast, our server code is written in Python 2.7. Both use the Google App Engine API [1]. In Section 3.1, Section 3.2 and Section 4 we discuss how we utilize Google App Engine in our implementation.

3.1 Client and Server Communication

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 Datastore and Blobstore

As mentioned before, Datastore and Blobstore are used to store a variety of information. Both Datastore and Blobstore have constant access time and are highly reliable means of storage. Google App Engine's Datastore uses the High Replication Datastore built atop BigTable [5]. Writes use the Paxos algorithm and changes are propagated to non-participating replicas asynchronously. App Engine Datastore provides high read/write availability, atomic transactions, and strong read consistency using get(). Queries, however, only guarantee eventual consistency. Thus, for our application, we use get() to retrieve entries for time-sensitive operations, such as propagating control commands from the host, and queries for less urgent actions. [1] [8]

Blobstore is a datastore that holds large data objects (blobs) that are too big to be stored in the Datastore. Once a blob has been created, it cannot be modified; thus, it can be considered read-only storage. We choose Blobstore to serve our content due to the higher file size limit. Moreover, it also performs automatic caching on its data. [1]

4. DESIGN CONSIDERATIONS

We made certain decisions in order to achieve the goals discussed in Section 1. In this section we summarize and discuss these choices.

1. **Performance**: Songs are fetched by the client as soon as they are available, allowing for seemless transitions between songs. We also perform synchronization, as discussed in Section 2.2.3, and use eventually consistent queries when possible, as mentioned in Section 3.2.

2. **Availability**: We rely on Google services such as Datastore and Blobstore, which are highly replicated and reliable. In addition, HTTP requests to the server are served as long as Google App Engine is available. We discuss how our system behaves with various failure modes in Section 4.1.

3. **Scalability**: For scalability purposes, we allow concurrent requests. To minimize the number and size of messages, we allow incremental updates. We do not send all session data on an update except when a listener joins a session.

4.1 Fault Tolerance

There are several components of our system that could fail. The host or listener of a session could lose connection to the server or the master/session server could go down. We discuss each type of failure below with respect to the impact on other users of the system.

When the host of a session goes down, the system mainly continues to operate as normal. Any listeners in the session of the down host will continue playing the current song until completion. At that time, they will wait until the host reconnects and updates the session server with new information. Other clients will still be able to join the session, but will have to wait like the current listeners. Once the host reconnects, it rejoins its hosted session and continues playback as if it just joined the session.

A listener losing connection has very little impact on the system. The session server will detect the channel failure and cleanup necessary state associated with the listener. Other listeners as well as the host of the session will not be impacted by the disconnected listener.

When the master handler fails, active sessions continue to operate as before. Hosts and listeners can continue to listen together to existing and newly added songs. However, new clients attempting to connect to the system will have their requests time out because the master handler manages all initial client connections.

If the session handler fails, all session updates will fail. Clients will finish the current song of their session and then will have to wait for the session handler to come back up. New clients first connecting to the system will be able to login, but will not be able to join a session. They will also be unable to start a session until the session handler comes back up.

Note, however, that Google App Engine uses multiple web servers to handle requests reliably and that requests can be handled by any of those servers. Moreover, the number of servers is automatically adjusted based on load. Thus, as long as there is at least one server available running our handlers the system will operate as normal.

5. EVALUATION

We evaluated our system on the metrics of performance and scalability. To measure performance, we designed benchmarks to measure the data rate of a listener versus the data rate of the host. For scalability, we measured system performance as we added clients to the system. We can achieve this by launching our application on an increasing number of client machines.

5.1 Performance

For the performance of our system, we were interested in differences in playback between clients in the same session. We wanted clients to be listening to not just the same song, but the same position in the song. To test this, we set up two different experiments.

1. Determine differences in playback between a host and a listener as more active sessions are added to the system.

2. Determine differences in playback between a host and a listener as more listeners are added to their session.

The host and listener were set up as different users on the same machine so as to have identical clocks. The added sessions or listeners were achieved through a modified version of Google's load test for App Engine applications. Differences in playback were observed through comparing the ending timestamps of songs between the host and listener. Each experiment followed a similar series of steps, listed below.

1. The host starts a session by choosing a song.

2. The listener joins the host's newly created session.

3. New sessions/listeners are added to the system.

4. The time the current song ended is recorded for both the host and original listener.

5. The host adds a new song. Go to step 3.

The results for adding more active sessions are shown in Figure 4. As more sessions were added to the system, there wasn't much of a difference in song playback between the host and listener. On average the difference actually decreased as more sessions were added, but this is likely due to the outlier observed for 30 sessions. The minor differences observed are more likely to have been caused by external network-related factors than by our system itself.

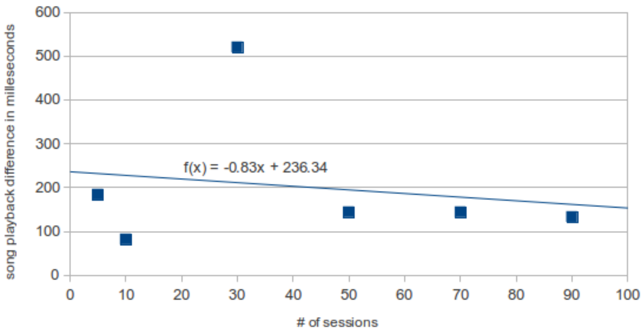


Figure 4: Playback difference when adding more sessions.

The results for adding more listeners are shown in Figure 5. As more listeners were added to a session, there was a minor increase in song playback differences between the host and listener. However, even with 90 active listeners in a session, the difference was less than 1 second. This increase is likely due to the additional work the server has to perform to propagate updates to an increasing number of listeners in the session.

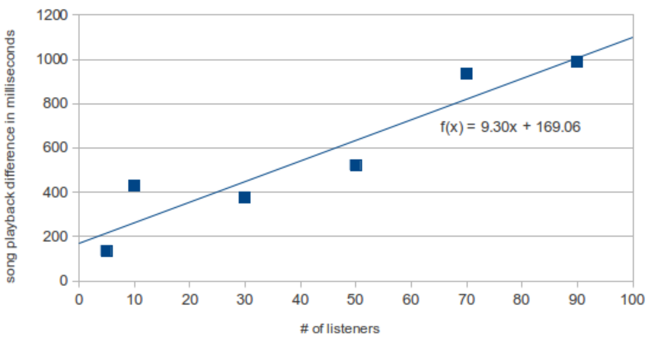


Figure 5: Playback difference when adding more listeners.

5.2 Scalability

In terms of scalability, we tested how the number of listeners in a session would affect the amount of time it takes for a new user to join. To test this, we had one client start hosting a session and then had another user join the session as a listener. On joining, it would time how long it took to recieve a response from the data server. A minute later, another user would join the session to time how long it took the data server to respond to its request. We repeated this up to one hundred users. We stopped at one hundred since we are limited to one hundred channels and each user requires a channel. As you can see in Figure 6, the first listener takes less than one second to join a music session whereas the the one hundredth user takes over four seconds to join a music session.

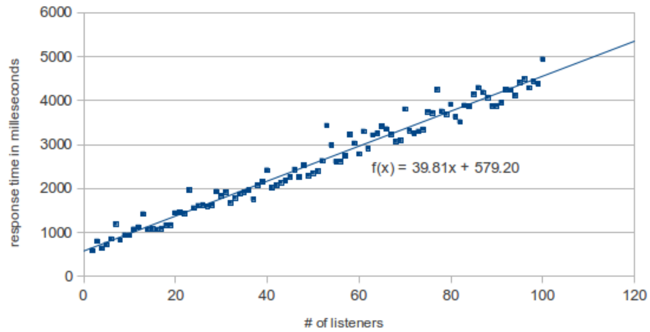


Figure 6: Time for listener to join a session

6. FUTURE WORK

Future work includes both adding features and improving performance.

Currently, our system allows any user to see any other user's session and listen to any music session. Ideally, users would be able to control who can see and listen to their sessions by creating access control lists (ACLs). Then, a user would only see the sessions to which they have access. While we have 'add' and 'remove' buttons that allow users to create ACLs, currently our system does not use this information to control sessions. The next step would be to use these ACL to actually control access. A nice feature to go along with this would be to allow users to 'search' for other users. The add and remove buttons only allow exact match of other user's email addresses. Another nice feature, like that in Google Hangouts [2], would be to allow users to invite other users to join a session by e-mail.

In terms of performance and scalibility, the easiest improvement we could make would be to use memcache in combination with the data stores. This could improve the response time for a client to join a session, since joining a session involves a lookup in a datastore for session and listener information. Another improvement we could make is to make incremental updates. For example, when a listener is removed from the listener list, instead of sending the entire list minus that listener, we could just send the listener that is being removed and let each client remove it.

7. CONCLUSION

We have developed a music sharing application called ClouDJ. ClouDJ allows multiple clients to listen to music together. A host can start a music session that other clients can join and listen to in realtime. ClouDJ uses a JavaScript-based client and a Google App Engine backed server. Through our design and implementation, ClouDJ is able to achieve our design goals of good performance, availability, and scalability. ClouDJ achieves good performance with many active sessions or listeners, supported by our evaluation results. It does so by using efficient synchronization between clients. ClouDJ also achieves high availability of music sessions through the use of Google App Engine services for sessions and songs. Additionally, it tolerates many component-based failures such as disconnected listeners or a master server failure. Finally, ClouDJ achieves good playback scalability as more clients are added to the system, shown in our evaluation section. It does so by using the highly concurrent and scalable infrastructure of Google App Engine.

8. REFERENCES

[1] Google app engine. `https://developers.google.com/appengine/`, Nov. 2012.

[2] Google hangouts. `http://www.google.com/+/learnmore/hangouts/`, Nov. 2012.

[3] J. S. A. Hittson, R. Lam. Cloudj application. `http://4.cloud-dj.appspot.com`, 2013.

[4] J. S. A. Hittson, R. Lam. Cloudj source code. `https://github.com/slauson/Cloud-DJ/`, 2013.

[5] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[6] G. Kreitz and F. Niemela. Spotify–large scale, low latency, p2p music-on-demand streaming. In *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*, pages 1–10. IEEE, 2010.

[7] S. Schiller. Soundmanager 2: Javascript sound for the web. `http://www.schillmania.com/projects/soundmanager2/`, Nov. 2012.

[8] J. Scudder. Life of a datastore write. `https://developers.google.com/appengine/articles/life_of_write`, Nov. 2012.