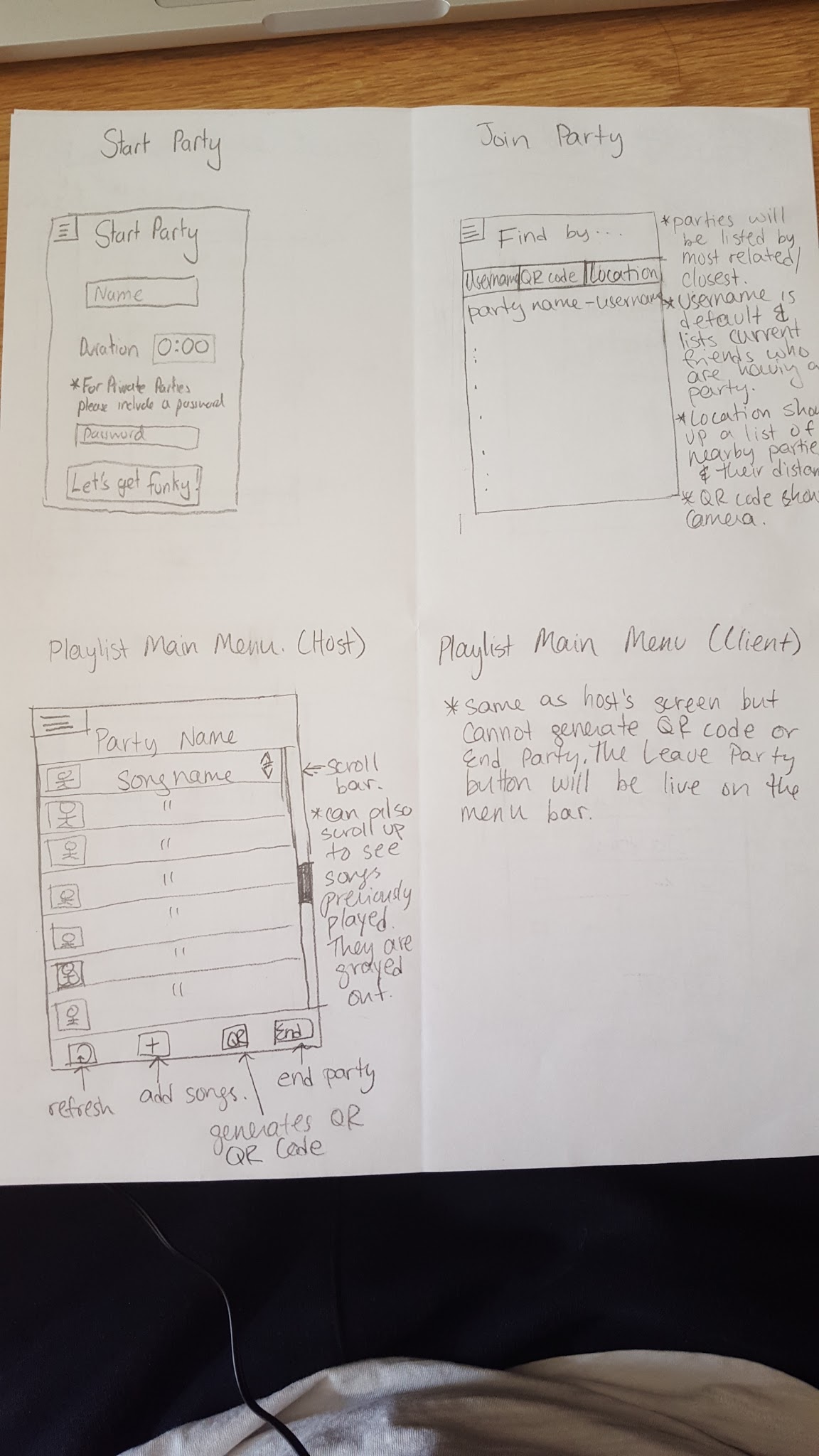
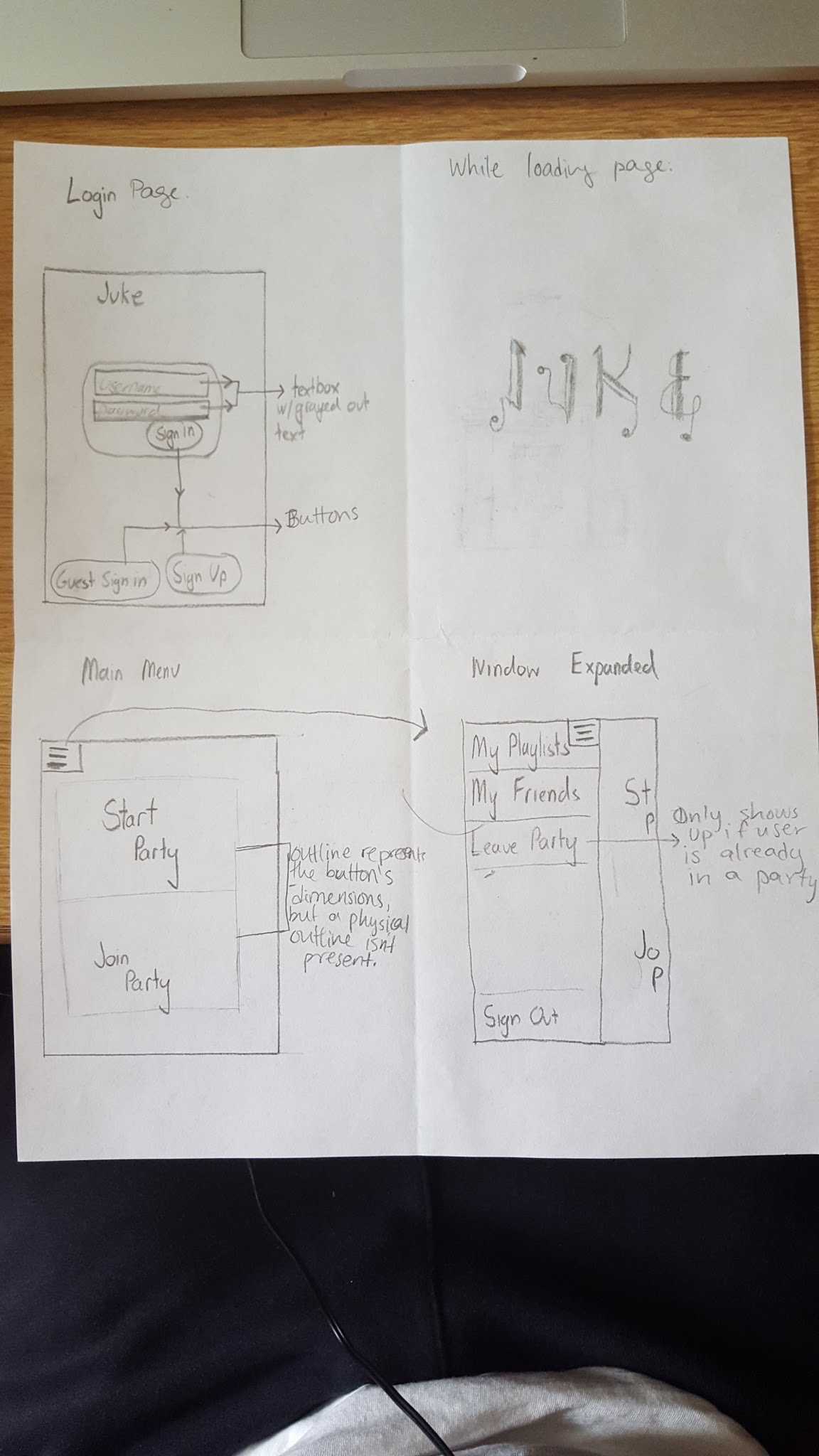
**Juke: Detailed Design Doc**

**Rama Gosula gosula@usc.edu 30393   
Meghan Mehta Mehtam@usc.edu 30393   
Chris Cognetta Ccognett@usc.edu 30393   
Jason Roodman roodman@usc.edu 30393   
Nayha Kamboj nkamboj@usc.edu 26361  
Katie Park Katiepar@usc.edu 30393**

**Front End:**



**Generating songs:**

When selecting songs, the app will generate a list of all songs contained in the android Media file (retrieved using the Android Media Playback API) and those will be the general options for music. Clients can check off songs they want to add, and if that song already exists in the list, then it will count as an upvote.

**Generating playlist:**

The playlist will organize based on the votes of the songs currently on the list. Scrolling up will show all previously played songs. A song can be re added immediately after it is played. These will pull from the played and toPlay arraylists described in the playlist class below

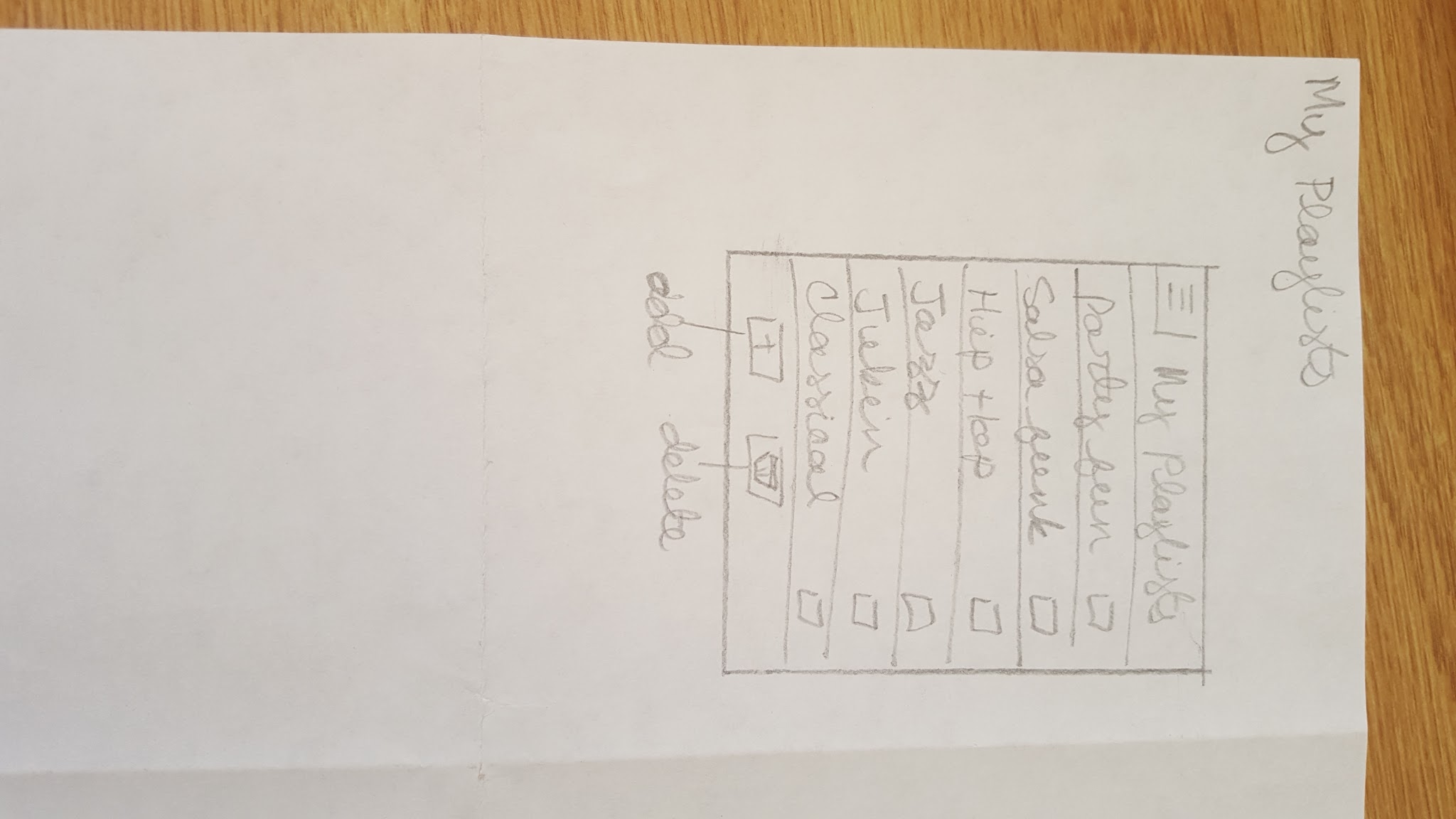
**QR Code:**

We can develop and scan QR Code via the QR Droid Zapper library.

**Location:**

We can use Google Play’s services location API to get the user’s last known location and compare all of the last known locations for all active users to compile a list of users that are nearby.





**Back End:**

Database:

The database consists of 4 tables:

User

|  |  |  |  |
| --- | --- | --- | --- |
| userID | username | email | hash |
| Auto generated, primary key, not null | Not null, unique | Not null | Not null  \*hashed password\* |

Friend

|  |  |  |
| --- | --- | --- |
| friendshipID | followerID | followedID |
| Auto generated, primary key, not null | fk(User.userID),  \*represents which user is following\* | Not null, fk(User.userID),  \*represents which user is being followed\* |

Note, the friendship feature works more like twitter’s follow system.

Playlist

|  |  |  |  |
| --- | --- | --- | --- |
| playlistID | userID | playlistname | filepath |
| Auto generated,Primary key, not null | fk(User.userID),  \*represents owner of playlist\* | Notnull, \*playlist name\* | Not null \* path where the playlist is stored on the server’s file system\* |

PlaylistContent

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| playlistSongID | playlistID | songIndex | songName | songArtist |
| Auto generated,Primary key, not null | fk(Playlist.playlistID),  \*represents specific playlist\* | Not null  \* Song’s position in the playlist \* | Not null  \* Song name\* | Not null  \* Artist name\* |

Networking Framework:

There are two server-client style interactions present.

1. That between the webserver and all connected clients. The webserver hosts the database and all of the saved playlists.
2. That between the host (or jukebox) device and all of the clients connected to it during playlist runtime

**Webserver**

For the webserver interactions, the client connects directly to the server through jdbc and queries the database through the container class:

|  |
| --- |
| DatabaseConnector |
| String hostname  Int port  String user, password |
| DatabaseConnector(hostname,port,user,pass)  Boolean userExists(String username);  Bool createUser(String username, String email, String password);  Bool login(String username, String password);//checks is login was successful  List<String \*usernames\*> getFriends(String username);//return friendlist  Bool followFriend(String currentUsername, String toFollowUsername);  Bool playlistExists(String playlistname,String username);  Bool createPlaylist(String playlistname, String username, String filepath)//save the playlist to the server’s filesystem and adds its info to the database  Bool copyPlaylist(String currentUserName, String SelectedUserName, String playlistname);  //copies a playlist from one user to another |

**Juke host to client connection**

Songs are stored in a wrapper object

|  |
| --- |
| Song |
| String localFilepath  String name  String artist |
| Song(String filepath) |

PlaylistSongs are objects stored within the playlist with extra information for a song.

They are elements of the playlist, so they need to store this info

|  |
| --- |
| PlaylistSong |
| Song song;  Playlist owner;  Int voteScore  Boolean played |
| PlaylistSong(Song song, Playlist owner)  Void upvote();  Void downvote();  Int getScore(); |

The running playlist is stored in a playlist object:

|  |
| --- |
| Playlist |
| Arraylist<PlaylistSong> allSongs;  Arraylist<PlaylistSong> played;  Arraylist<PlaylistSong> toPlay;//this arraylist will be updated throughout playback  Arraylist<PlaylistSong> toPlayBuffer;//this will store the updates to the playlist and is swapped with toPlay when it is all complete.  PlaylistSong current; //this is the song currently playing |
| Playlist()  Bool addSong(Song)  Bool Upvote(PlaylistSong)  Bool downvote(PlaylistSong)  Void sortToPlay();  Song advanceSong() // advances to the first song in the to play list, setting current, removing it from the to-play list, and adding it to the played list. The current song is updated then returned as a Song |

**Communication**

**Client to Host**

For the jukebox, communication is accomplished through ServerCommands (also known as Command Objects), which are sent through socket and executed on the server once they are read into a ServerCommand object.

ImmediateCommands are also used for interaction, and are used to be executed immediately upon receipt on the Host.

The host also interacts with the playlist through the commands, in order to simplify execution.

|  |
| --- |
| (Abstract) HostCommand |
| PlaylistSong song  Client client; |
| ServerCommand(Song, Thread client);//the song is the song on which an action will be peformed, the client is included so that the host can reply to the client that invoked the command  //TODO What actually represents the client?  Abstract void execute(Playlist); //this will execute based on the type of command; |

Subclasses of Server Command:

[AddSongCommand] extends ServerCommand

//execute adds the song to the playlist in the arguments

//if already present in the playlist, it upvotes the song

[RemoveSongCommand] extends ServerCommand

//execute removes the specified song if in playlist

//note: only the host should be able to execute this command

[UpvoteCommand] extends ServerCommand

//upvotes song if it exists

[DownvoteCommand] extends ServerCommand

//downvotes song if it exists

|  |
| --- |
| ImmediateCommand |
| Client client;  String label; |
|  |

ImmediateCommand has a siimilar purpose to a ServerCommand, but does not consist of an operation on a specific song and an execution function, instead it contains a label of which operation to perform, and which client sent it. It is sent through the same stream as ServerCommands.

Labels:

“GetAvailableSongs”

//requests the host to send over its list of songs that can be played

“RefreshPlaylist”

//requests for the host to send the playlist object currently connected to so the client may refresh its version of the playlist

**Host to Client**

The server will reply to the client with a ServerReply object

|  |
| --- |
| ServerReply |
| String label;  Object data; |
|  |

The label indicates which kind of data is sent:

“Boolean”, “Playlist”,”SongList”

A boolean object is sent in reply to any of the commands that operate on the playlist to confirm successful operation.

A Playlist object is sent in reply to RefreshPlaylistCommand to be used to refresh the playlist.

A SongList object is sent in reply to GetAvailableSongsCommand so that the client can populate their menu to add songs

**The Client’s Threads**

During runtime, all the clients are running the following threads. Reminder: host behaves like a client

|  |
| --- |
| CommandThread |
| Queue<ServerCommand> commands;  ObjectInputStream ois;  ObjectOutputStream oos; |
| CommandThread();  Void addCommand(ServerCommand);  Bool executeCommand();  Void run(); |

CommandThread.run() is an endless loop executing the popped instruction.

Every popped instruction waits on a ServerReply to be sent back. If the command does not execute, successfully, it loops again, insuring that every command is executed.

If a RefreshPlaylistCommand is received, the client’s playlist is instantly refreshed

|  |
| --- |
| ClientRefreshThread |
|  |
| ClientRefreshThread()  Void run(); |

ClientRefreshThread.run() just calls addCommand(new RefreshPlaylistCommand()) then sleeps so that the playlist refreshes every couple seconds.

**The Host’s Threads**

These threads only run on the Host or Jukebox machine. They run during the playlists existence

|  |
| --- |
| UpdateThread |
| Queue<ServerCommand> commands;  Playlist playlist;  Bool updating;  ObjectOutputStream oos; |
| UpdateThread(Playlist);  Void addCommand(ServerCommand);  Bool isUpdating();  Void run() |

The Update thread receives commands. On run, it has an endless loop that executes all of the ServerCommands in the Queue on toPlayBuffer, sends the proper ServerReply, and then sorts the playlists’s ToPlayBuffer list (since PlaylistSongs will have changed score, or have been added or deleted), and copies it to toPlay;

While executing the commands in the queue, updating is set to true, to prevent access of the playlist in mid-operation.

|  |
| --- |
| ReceiverThread |
| UpdateThread updater;  Host host;  ObjectInputStream ois; |
| run(); |

On run(), ReceiverThread has while(true) loop that reads ServerCommands or ImmediateCommands.If it receives an ImmediateCommand, it will read which type it is and execute it immediately, sending the data back to the client in a ServerReply.

If it recieves a ServerComand, it passes it to the UpdateThread through addCommand();