Brandon Vickrey
Michael Lee
Ryan Schneider
Austin Rauschuber
Joe Swanson
Mitchell Hanberg

Design Document (Group 9 / Blotto Beats)

1. Purpose:

There are many companies that need music for their projects but are either unable to afford that cost or spend that time. We would like to create a program which can procedurally generate music for these types of individuals. Not only could our procedurally generated music help these people, but also could help others such as people studying music theory, musician hobbyists, and people that just want to hear some music.

2. Non-Functional requirements:

Generation Time: As a user, I would like tracks to be created in a trivial amount of time. This is because the client serves no purpose to the user during the song generation time, so the time should be trivial at most.

Client Stability: As a user, I would like the application to be stable over long periods of time. This is because the application is designed to be able to run continuously in the background over larger periods of time, and stability is necessary for the client to perform this basic function.

Generation Variance: As a user, I would like there to be sufficient variance within the random tracks. This is because music serves the purpose of being interesting both from a compositional and listener standpoint when there is variation between songs.

Server Resources: As a developer, I want the server to be lightweight on resources. This is because we will have reasonably limited server resources, and there will be large quantities of data to be handled.

Modification Time: As a developer, I would like the addition of new generator parameters to have trivial modification time. This is because song variance and interest is increased with the number of potential parameters, but not all parameters are possible to insert at one single point in time. For example, different genres have drastically different compositional structures that will be added at separate times.

Server Data Set: As a user, I would like the server to be able to handle the input of thousands of track data sets. This is because the program will be able to generate massive quantities of different tracks, and the server should be able to handle the voting information on as many of these tracks as possible to provide the info to the user.

Project Reusability: As a developer, I would like to be able to incorporate the generator algorithm into other projects easily. This is because the usage of music is very common in a wide variety of programs, and this procedural generator could be used in other projects for music composition.

3. Design Outline:

We're choosing the client-server model. We will be implementing a fat client, which will do the majority of the work, including letting the user input variables, generating a song for the user, saving the variables with the song. The client will store the user's personal voting records also. The server will store songs that the chooses to upload, as well as storing the voting information for tracks, and the variables for the track that they are connected to. The voting system will just be a score attached to each song. The way we plan on storing the songs in the database is to place the songs in buckets first based on the genre parameter of the song then from there we will put those songs in buckets based on the seed. There could be several songs with the same seed, but with the other parameters being different, it will produce a different song. This is why we feel sorting by first the genre then by the seed is the best way to store the data.

The generator will generate the song when the play button is pressed in the client.
The song will play when the client gets a message from the generate function that the song has been generated. The song will be locally downloaded. When the user presses the upvote or downvote button, the client will send a message to the server, if the song is in the database, then it will update the votes, if not, then the song will be added to the database.

4. Design Issues

Song Ranking Storage
Store on the server
Store on the client

We chose to store upvotes and downvotes on the server as that was where the songs that are shared get stored.

Song Ranking Format
Store individual number of downvotes and upvotes
Store one overall score

We stored it as one score in order to make handling the data on the server easier, as there was no need to separate the two values.

List of Previous Voters
Store on the server
Store on the client of the voter

We stored the data on who has voted for a song on the clients, as it is simpler to manage and it saves space on the server.

Song Grouping
No grouping
Grouping by genre
Grouping by one or more of the parameters

We chose to sort the songs by genre, as that was a simple way to divide up the songs in a logical manner

Song Indexing
Use seeds as index
Assign a unique ID
Create a hash

We chose to index the songs by the seed, as there would be almost no collisions between songs with the same seed. Any songs that do will be further indexed by their other parameters.

5. Design Details:

Our system will consist mostly of three classes. We will have the client, which will be the main window that the user will be using to interact with the system; the generator, which will be where we will be generating the music; and the server, which is where we will be storing the upvote/downvote score for all of the songs.

Client
Most of the end-user interaction will be based in the client. The client handles all of the GUI and most of the preferences and settings. Whenever the client requests a new song (e.g. by pressing the play button or having autoplay enabled) or votes on a current song, the client will interact with the server and the generator.

When the client requests a new song, it will first request the master list of generator settings from the server. The client will combine those settings along with a local list of generator settings, to create customized list of generator settings. It will then pass that list -- along with a seed and the user's current song parameters -- to the generator, which will return a song to play.

When the user votes on the song, it will send the song's seed and parameters, along with the vote, to the server. The server will add the song to its list if it is not already there, and will add the user's vote.

The user will also be able to request lists of songs as well. The client will send a request to the server with the user's search parameters, and will display the list of the songs that the server sends back.

Generator
The generator's purpose will be to take data presented in the form of parameters and a seed from the client and output a song in a playable format. This will take the form of a series of methods and data structures.

The seed will be a number that is utilized in generation to create constant outputs by our random generator. The outputs of our random generator will determine things such as key, chord patterns, overall song patterns, etc.

The generator class will have a generateSong() function. This function will be the function called from the client and its parameters will be a seed, a GeneratorSettings object and a SongParameters object. The seed and GeneratorSettings will be fed to a random number generator. Using the output of that along with the SongParameters object, the generateSong() method will then utilize a deterministic generation algorithm to create an instance of the Song class.

The Song class will contain a digital representation of a musical composition. This will include things such as the list of chord progressions and melodic structures.

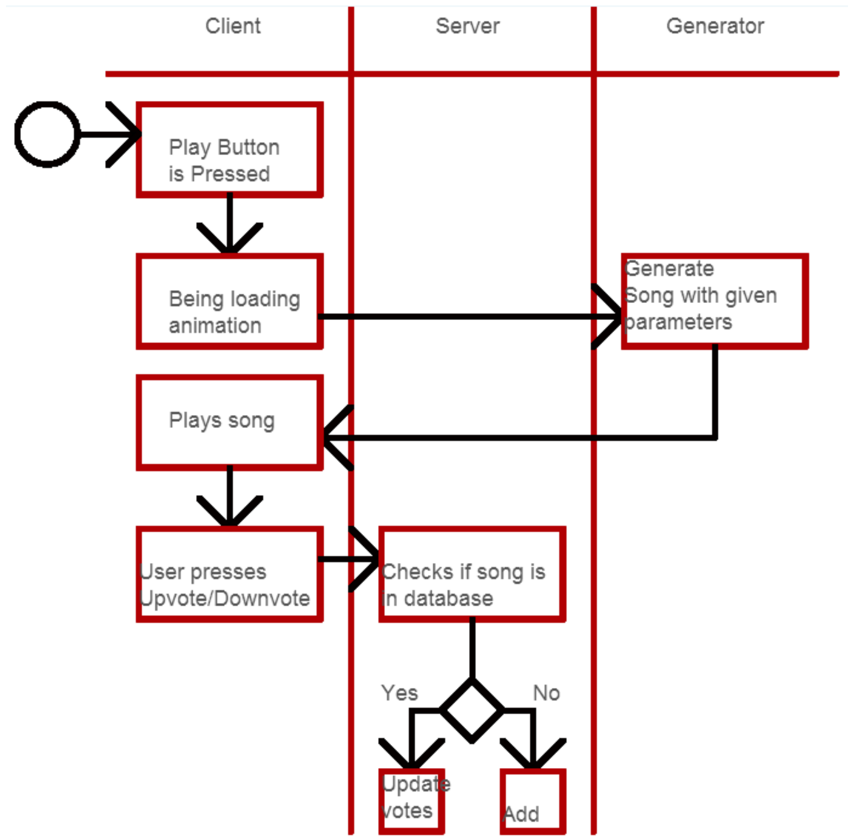
Prior to termination, the generateSong() method will call the parseSong() method. The parseSong() method will take the data stored in the instance of the Song object and parse it into a format playable by the client. This will most likely be a MIDI format. The parseSong() method will then return the location of the MIDI file to the generateSong() method.

Server
The server's purpose is to serve as a central repository for songs. If a person likes or dislikes a song that they generate, they will up- or downvote the song, which will check the server for that specific song. If it exists, the vote will be added to that song's score. If it does not exist, the seed, genre, and parameters used to generate that song will be uploaded.

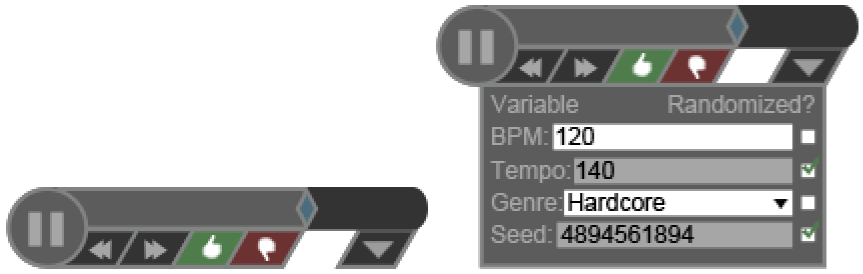
The client will be able to request a list of popular songs based on specific preferences, such as genre, beats per minute, length, and whatever other parameters we find appropriate. The server will send a list of the most popular songs that match those parameters, and the user is free to listen to and vote on those songs.

The server will also be responsible for storing a master list of global seed-generator settings. Every time a client generates a song, it will request the server's list of settings, which it will modify using the local user's preferences and used to generate the new song. If time permits, we will add a machine-learning component to the song generator. With that enabled, the settings will be modified based on which songs are most and least popular.

Example interaction flowchart:



Mock UI Design:



Variable		Randomized?
BPM:	120	☐
Tempo:	140	✓
Genre:	Hardcore	☐
Seed:	4894561894	✓