Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Listening to Music in Groups

Semester Thesis

Gino Brunner

brunnegi@student.ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Jara Uitto, Barbara Keller
Prof. Dr. Roger Wattenhofer

July 30, 2014

Acknowledgements

I would like to express my gratitude to Prof. Dr. Roger Wattenhofer for the opportunity to write this semester thesis in his research group.

I also want to thank my supervisors Barbara Keller and Jara Uitto for the constructive and fun collaboration that made this thesis possible.

Abstract

We evaluate the music taste representation computation and song matching capabilities of Jukefox, a smart music player for Android developed at the distributed computing group at ETH Zürich. We then use this music taste representation for single persons to create group playlists. A group playlist contains the aggregated songs of all group members, ordered from best to worst. The more group members like a song, the better it is for the group.

We developed an application that uses the group playlist-concept to allow a group of people to automatically set up a listening session, where the application always plays the best songs for the group. The users can easily join the group by pressing a button, which connects the android phone to the server that runs a desktop client. The server communicates with the clients over (W)LAN to request their songs and music tastes. The server then computes the group playlist and requests the best songs from the clients. Once the server has received the mp3 files, it starts to play them. Clients can join and leave at any time; the server then automatically computes the playlist for the new group and, if necessary, requests new mp3 files.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

It's Friday night again. You and your friends are sitting in your favourite bar, having a good time. The liquor is already flowing like water. The only thing that could make the start of the evening even better is good music. Unfortunately there is no DJ tonight, and all that's on is the most generic mainstream noise, which will all too soon make you leave for another place in search of music that suits your group. Sounds familiar?

What if there was a way to have bars play the music that best suits the crowd? This is where the concept of `Music Taste Representations`, or rather `Group` Music Taste Representation, comes in. If we know a group's music taste, songs could be played accordingly, and everyone would be happy. No matter if a bar is filled with metal-heads, k-pop fans or friends of classical music, the group music taste would automatically cater for their specific needs on any particular evening.

This thesis aims at providing a first proof of concept towards above described goal. We show that Jukefox' music taste representation can be used to create play lists fit an entire group. Additionally, we provide a functioning implementation that show cases our findings.

Todo: Expand a little bit, state more clearly what the goal is, i.e. working implementation blabla.

1.2 Outline

The remainder of this thesis is structured as follows:

Chapter 2 gives an overview of prior work concerning music tastes from different areas of research.

Chapter 3 introduces Jukefox and its music taste representation. We also explain how the music taste concept can be extended to groups of people, and

how we actually use the music taste representation to create group playlists, which are then used in the application.

In Chapter 4 we present several experiments that allowed us to get a better understanding of the existing music taste representation. We evaluate the group playlists and show how much better they are than randomized song-picking. We also assess the accuracy of the music taste computation, and explain the implications for this thesis.

Chapter 5 shows the finished product. We explain the architecture and features of the application, and show how it is used. We give a detailed overview of the application's communication protocol,

Chapter 6 presents possible directions for future improvements and further research.

Chapter 7 brings the thesis to a conclusion.

Chapter 2

Related Work

The problem of music-taste based music-retrieval has been relevant for quite some time. One prominent, recent example is its application in Spotify, where new music is proposed based on what you have been listening to. This section gives a short overview over the different research directions for music recommendation and music tastes.

Many papers also address music taste from a psychological and behavioural-science point of view, which might give valuable insight into the topic. Alicia Dunning ([1]) conducted interviews with people from two different classes to find out whether they have different music tastes. Juul Mulder et al. [2] expand on several research papers that compare behaviour of people liking main-stream music versus those preferring "deviant" music (such as rap and heavy metal. They performed a clustering analysis on the music preferences of youngsters and found seven distinct music taste groups. Such results are interesting from a research point of view, but since they are not quantitative, cannot be used directly for music recommendation.

There have been different approaches to define music tastes, or to find music that a person likes (Music Information Retrieval MIR), which basically defines music taste in an implicit way. There are mainly two (complementary) approaches for MIR: (1) collaborative filtering and (2) content-based retrieval. Furthermore, music similarity can be defined by tagging (either collaborative, by experts or through a labeling game), by comparing music piece profiles to user profiles, or by automatic signal analysis. The latter yields information about timbre/instrumentation, harmonic content and/or rhythm [3].

Foote [4] computes the similarity between audio files (e.g. songs) based on templates derived from a supervised tree-based vector quantizer (`TreeQ`). Hoashi et al. [5, 6, 7, 8] developed content-based music information retrieval (MIR) methods based on the `TreeQ` algorithm devloped by Foote. The above approaches (since based on `TreeQ`) have to be trained by human input. A sample of songs must be rated by users to establish a baseline for the algorithm. Other papers in the area of content-based MIR are by Logan [9, 10] and Grimaldi et al. [11].

In [12], Kuhn et al. introduces a high dimensional music similarity space based on user driven similarity measures. The paper is the base for the `Jukefox` music player music taste computation, on which this thesis relies. Jukefox uses the methods and algorithms developed in [12] together with user-data from last.fm to create a 32-dimensional music similarity space where each song has a unique position, and based on that, a user's music taste is computed by applying k-means clustering to his song collection. This music taste representation can then be used to find new music that the user likes.

Chapter 3

Jukefox and Music Taste Representation

This chapter shows how Jukefox's music taste representation is used to compute a play list that is optimal for a group of people.

Section 3.1 gives a short overview of Jukefox, concentrating on the aspects and features that are relevant for this thesis. Then there will be an introduction about the way Jukefox computes and represents the music tastes of users in Section 3.2. Finally, in Section 3.3, we discuss how individual group members' music taste representations are used to create a playlist that best suits a particular group, which also implicitly defines a `group music taste`.

3.1 Jukefox Introduction

Jukefox is a smart music player for Android developed at the Group for Distributed Computing at ETH Zurich. Its goal is to make users happier by playing songs in a "smart" way. It incorporates explicit metrics such as ratings, as well as implicit metrics such as skipping a song, which reduces the likeliness of it being played again. When a user skips a song, Jukefox tries to play a song that lies within a "different area" of the person's music taste. By doing this, Jukefox also caters to the user's mood to a certain extent.

In order to do this, Jukefox uses a 32-dimensional `music similarity space` (MSS). Each song has a distinct position in the MSS and Jukefox can therefore compute the distance between two songs, which is used as a similarity measure. The closer two songs are, the more similar they are. The coordinates of songs are computed and stored on a server and can be retrieved by Jukefox. The MSS is constructed from user data on `last.fm`.

Within the MSS, the music taste of a user is represented by clusters. A song is defined to fit a persons music taste when it lies within one of those clusters, and vice versa.

In order to construct a user's music taste, i.e. compute the clusters within the MSS, k-means clustering is applied to all the songs in the user's collection. The songs are clustered and outliers are removed. In the end we get the music taste representation of a user based on his song collection. Therefore, in order to get an accurate representation, the user's collection shouldn't contain too many songs that the user does not actually like.

3.2 Music Taste Computation

Within Jukefox, the music taste of a user is represented by the SimpleMusicTaste class. When creating an new SimpleMusicTaste object, we need to pass an array of tuples, consisting of a 32-dimensional coordinate vector and a weight, for each song the user has in his collection. The weight allows us to influence the k-means algorithm by specifying the importance of individual songs. In this thesis we did not have any metric to distinguish the importance of songs at the time of music taste creation, therefore the weight is set to one for all songs.

After the creation of the SimpleMusicTaste, there are three important attributes that unambiguously define the music taste:

- Cluster center coordinates
- Cluster radii
- Number of clusters

It should be noted that the k-means clustering is not deterministic, since it chooses random starting points. This means that each time a user's music taste is computed, the result might be a little different. This should, however, not impact the "quality" of the music taste. It would anyway be difficult to assess the quality of different music taste representations of a single person, therefore we do not take this into further consideration.

Once the music taste is computed, we can get the similarity rating of a song, i.e., the distance to the closest cluster's center. If that distance is smaller than the radius of the sphere, i.e., the song lies within the cluster, we say that the song fits into the music taste of the user.

Combined with the AdHoc sharing capabilities of Jukefox, this can be used to expand one's music collection by letting Jukefox find suitable songs from other user's collections and transferring them to one's own device. Section 3.3 shows how this concept is expanded to create group playlists.

Figure 3.1 shows the concept of music tastes in Jukefox. User 1's music taste is represented by the dotted circles (clusters). All the songs that lie within one

of those circles fit user 1's music taste. If user 1 wants to find new songs that he likes he can request all songs from user 2 that lie within one of his clusters (the dotted circles).

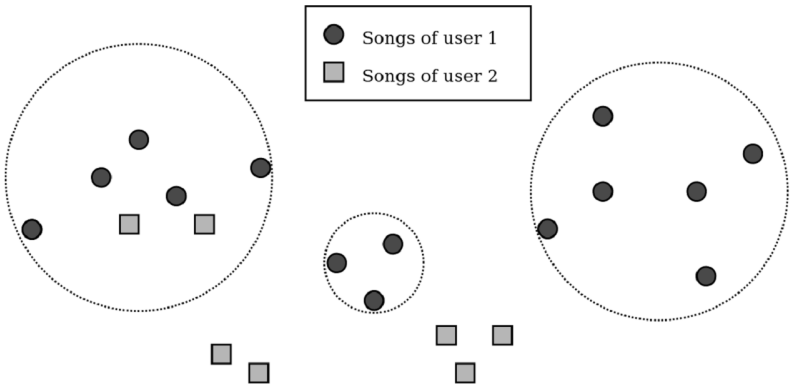

Figure 3.1: Visualization of a person's 32-dimensional music taste in 2-D [13].

3.3 Listening to Music in Groups

One of the goals of this thesis is to use Jukefox's existing music taste representation to create playlists that are optimal for groups of people. The scenario, as outlined in the introduction, is that two or more people meet and want to listen to music together. Each user contributes his music taste representation and his songs, and the application then creates a playlist for this group and allows them to directly listen to those songs.

3.3.1 Group Music Taste

In theory, a group music taste could be created by computing the parametrizations of all intersections of the individual users' clusters. By doing so, one would exclude songs that only a single person likes, and include all songs that at least two people like. However, without additional computations, there is no distinction between different cardinalities of intersections, e.g. two-fold and 3-fold intersections are treated the same.

Figure 3.2 shows the intersections of the music tastes of three users. The resulting group music taste would consist of all intersections. Songs that lie within a 3-fold intersection should be treated differently than those that are only liked by two users, since they satisfy all group members. This can be done by checking a song against the music taste of each group member, in addition to checking it against the group music taste.

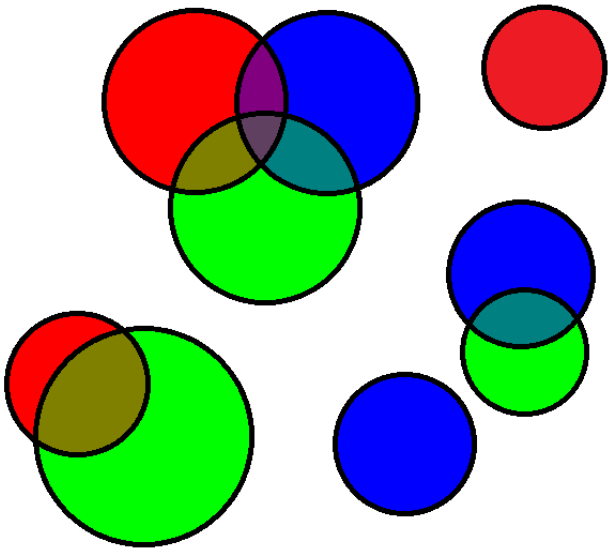

Figure 3.2: Visualization of group music taste with intersections of different cardinalities.

By explicitly computing the music taste for a specific group, new songs fitting the group's taste could very easily be found and new users' music tastes could directly be merged into the group music taste, because the parametrization of the group music taste can be used in the same way as music tastes of single persons. Furthermore, the group music taste representation could be stored and shared just like the existing `SimpleMusicTaste` objects.

This theoretical approach differs from the actual implementation described in section 3.3.2, where the group music taste is only implicitly computed in form of a `group playlist`. The advantage of this is ease of implementation and the possibility to directly distinct between n-fold and n+1-fold intersections.

3.3.2 Implementation as Group playlist

As mentioned in section 3.3.1, the actual implementation differs from our theoretical idea of a group music taste. Since this thesis presents a first proof of concept, the main reason is ease of implementation. It could be subject of future work to determine whether the explicit computation of a group music taste representation brings any real advantages (also see section 6.2).

The following algorithm shows a simplified version of the playlist computation:

Algorithm 1 Group Music playlist Computation

```
songs = getAllUsersSongCoordinates()
musicTastes = getAllUsersMusicTastes()
for Song s in songs do
  for MusicTaste t in musicTates do
    if t.getRating(s)≤1 then
      s.likeCount++
    end if
  end for
end for
Sort songs by likeCount
```

The result of the above computation is a list that is lexicographically ordered by `likeCount` and `ownedCount` (not shown in the algorithm). LikeCount indicates the number of people that like a song, i.e. the song lies within their music taste. OwnedCount indicates how many people have the song on their device. At the same time we also record `which` user owns the song, so we know from whom a song can be requested.

The playlist includes `all` songs of all users, even those that only have one like. This ensures that the computation also works for a single person, and it is easy to drop certain songs from the list anyway. The best (i.e. most-liked/owned) songs are at the beginning of the list.

3.3.3 Karma for Individual User Satisfaction

The group music playlist generated as described in section 3.3.2 ensures that we play the best songs for the group. As long as all users like the next song (i.e. likeCount equals group size), all users are "happy". If at some point we start playing songs with fewer likes than the group size, certain users might get dissatisfied because they don't like the songs being played. The karma concept ensures that individual users cannot get too unhappy. Each time a user has to listen to a song he does not like, his karma increases by one. As long as no user has a too high karma, we just play the songs top-down, i.e. best to worst according to the computed playlist. Once a user's karma reaches a certain threshold, we choose a song that this user likes, thus resetting his karma.

Chapter 4

Preliminary Experiments and Analysis

This chapter describes some experiments that were conducted in order to get a better understanding of the music taste computation in Jukefox and gain important insight before moving on to the actual implementation of the group music functionalities within Jukefox.

Section 4.1 describes the experiment setup and the preliminary steps that were to be performed prior to the actual experiments. Section 4.2 shows the first experiment where we examined the constitution of the group playlists, i.e. we wanted to find out how many songs are liked by how many users on average. This experiment was performed with real user-data. In Section 4.3 we try to get a better understanding of the results of the first experiment by performing it again, but this time with artificial users, leading to a more controlled experiment. To get a better idea of the nature of the Music Similarity Space, the experiment in Section 4.4 gives a rough idea about the density of the MSS, i.e. how much of the space is actually occupied with hyper-spheres. Section 4.13 goes a step further and gives a notion about the distance and overlap between two users' clusters. Finally, Section 4.6 discusses the outcomes of the experiments.

4.1 Setup and Preparation

Since Jukefox has been in the Google Play Store for quite some time, it has gathered a lot of information and usage statistics, which are all dumped into a database, together with data that was retrieved from LastFm. The database stores information like coordinates of songs, artists and albums, a play log, and also information about the users. This allowed for experiments with data of real users, which allowed us to simulate large-scale, real-world experiments.

In order to perform the experiments, "suitable" data was extracted and dumped into new database tables for the experiments:

Suitable Users: A table with suitable users. We decided to choose users that have at least 100 songs in their collection. We ended up with 141 users.

Suitable Users Songs/Artists/Albums: For all suitable users, tables representing their music collections were created.

Music Taste Representations: We computed and stored the music tastes for all suitable users in order to speed up the experiments. Reading the music tastes from the database is significantly faster than computing them. The music tastes were computed with no limit to the number of clusters for the k-means. Having many clusters should make the music taste representation more accurate.

Artificial Users: For the experiment in Section 4.3 we created artificial users that have very specific, artificial music tastes. This was achieved by creating users that have only songs that belong to one specific music genre. More information is given in Section 4.3.

4.2 Users vs. Songs Liked with Real Users

In this experiment we computed the group playlist for random groups of different sizes, while also varying the `acceptance threshold`. The acceptance threshold indicates how close a song must be to the center of a hyper-sphere to be considered to fit a person's music taste. Usually this threshold is one, which means that a song must lie within the sphere to be considered suitable. However, it is interesting to see how the threshold influences playlist computation. For each set of parameters, the experiment was repeated several times and the mean of all runs was calculated. For each run, the group members were chosen randomly from all suitable users.

The experiment was performed with the following parameters:

Group Sizes: $[2, 3, ..., 14]$

Acceptance Thresholds: 0.4, 0.6, 0.8, 1

Number of runs per parameter set: 20

Todo: Should i insert a plot with lower acceptance thresshold?

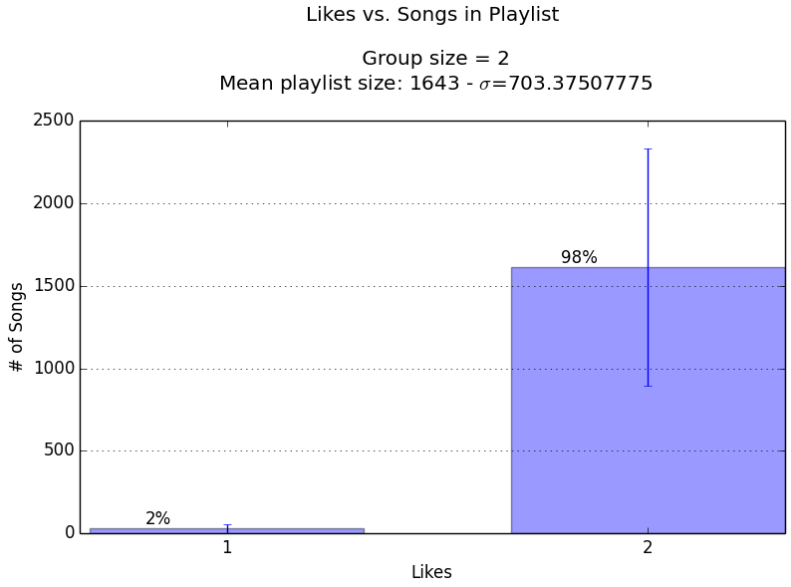

Figure 4.1: Like-distribution in the group playlist for two users.

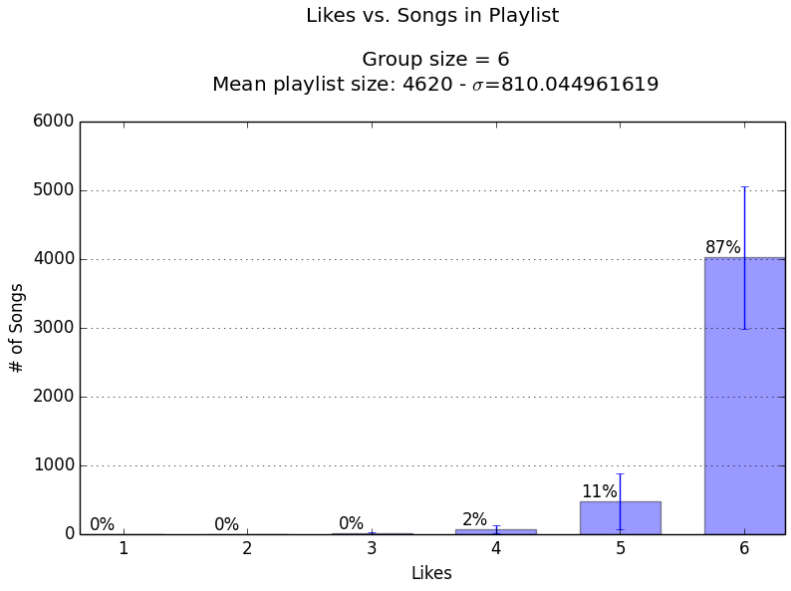

Figure 4.2: Like-distribution in the group playlist for six users.

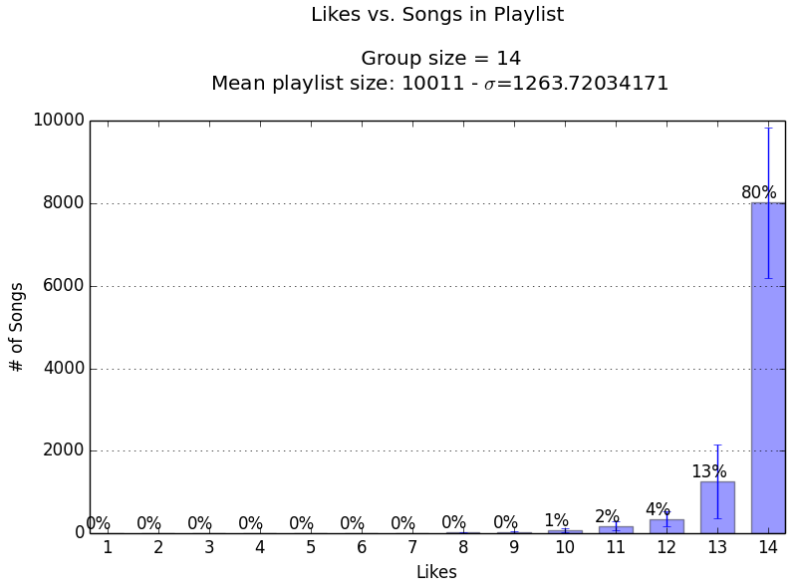

Figure 4.3: Like-distribution in the group playlist for 14 users.

The Figures 4.1, 4.2 and 4.3 show that there seems to be substantial overlap between users' music tastes. For all group sizes, a large portion of the songs are liked by many or even all users. Only when we drop the acceptance threshold are there fewer songs liked by all group members. However, dropping the acceptance threshold is more of a sanity check than intended for real use, because the entire music taste computation and song-matching algorithm is based around an acceptance threshold equal to one.

4.3 Users vs. Songs Liked with Artificial Users

This experiment is essentially the same as the one in Section 4.2, but instead of real user data, it is performed with artificially crafted users. This experiment is a sanity check to make sure that the music taste computation, and the extension to group playlists, is working properly. The users were created in such a way that their music tastes are reasonably and predictably disjoint. This should result in playlists where much fewer songs are liked by many people, i.e. the plots should shift to the left compared to the previous experiment.

For each artificial user one artist was chosen that represents a certain genre. We then put as many songs as possible of this artist into the users music collection. The created users look as follows:

1. Hip Hop: 2Pac

2. Rock: AC/DC
3. Classical: Franz Schubert
4. Electronic Music: Deadmau5
5. Country Music: Johnny Cash
6. Jazz: John Coltrane

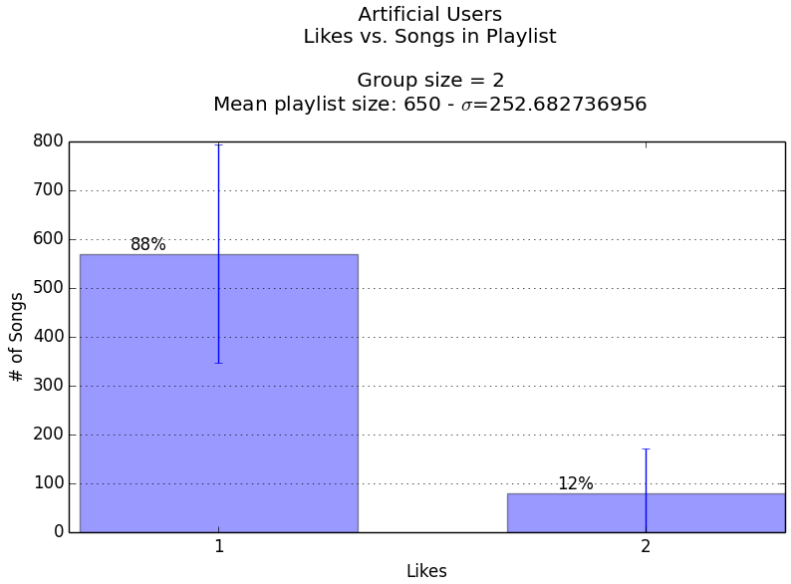

Figure 4.4: Like-distribution in the group playlist for two artificial users.

Figure 4.4 shows that for our six artificially crafted users there is roughly a ten percent pair-wise overlap between their music tastes. Or more precise: When comparing two users, ten percent of all songs (their combined song collections) were liked by both.

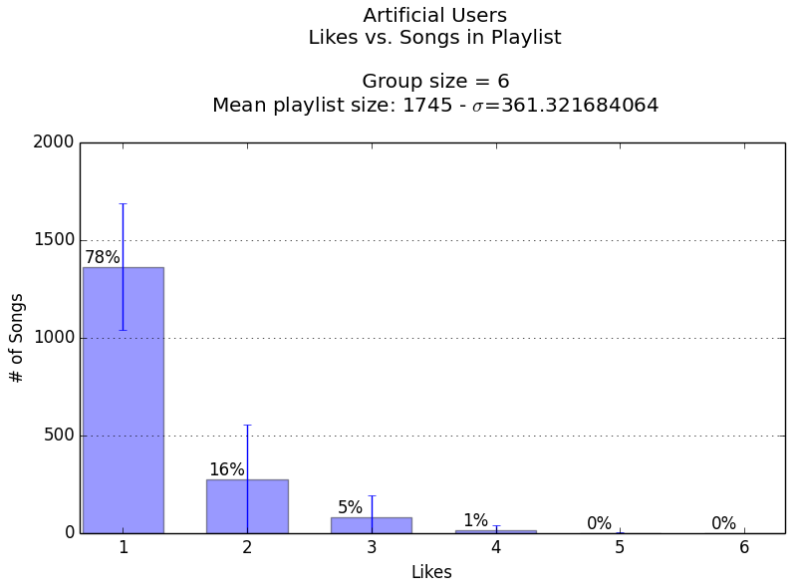

Figure 4.5: Like-distribution in the group playlist for six artificial users.

Figure 4.5 shows the results for a group consisting of all six artificial users. We can clearly see that the shape of the plot is the direct opposite to the plots in section 4.2. Almost all songs are only liked by one person (the owner of the song).

4.4 Hyperspace Density

When dealing with a 32-dimensional space, it is difficult to "know" what exactly is going on. One has to trust in the algorithms and that they indeed do what they are supposed to do. In order to get a clearer picture of the hyper-space and to better understand the results from sections 4.2 and 4.3, we attempt to compute the density of the MSS. In other words: How much of the space is occupied by the clusters that make up the music tastes.

In order to do so, we compute the volume of the hyperspace that encloses all clusters and divide it be the sum of all volumes of the individual clusters.

Volume of an n-dimensional hypersphere for high dimensions ($n > 10$): [1]

$$V_n(R) \sim \frac{1}{\sqrt{n\pi}} \left(\frac{2\pi e}{n}\right)^{n/2} R^n \tag{4.1}$$

We use formula 4.1 to calculate the volume of the individual spheres.

[1] Source: http://en.wikipedia.org/wiki/Volume_of_an_n-ball#The_volume

For the volume of the entire hyperspace we envision a large hyper-sphere that encloses all clusters. The diameter of this "super-sphere" is the maximum distance between any two cluster-centers, plus their respective radii.

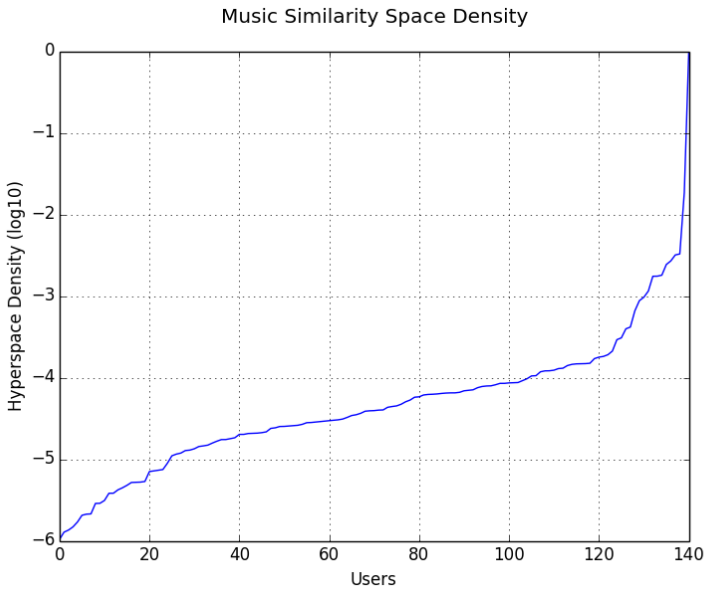

Figure 4.6: Hyperspace densities for all 141 real users. The values are taken to the logarithm of base 10.

Figure 4.6 gives an idea about the nature of the hyperspace. The fewer clusters the higher the density, with the extreme of a single cluster where the density equals one. If there are a lot of clusters, the density can get very small, especially when the clusters are very scattered, i.e. we have a `diverse` music taste.

However, it should be kept in mind that using an all-enclosing hypersphere is likely not the optimal way to do it, because one introduces a lot of `empty space` at the edge of the sphere. Experiment needs to be refined in order to get more precise and more usable data. This first attempt aimed at giving a rough idea about the magnitudes of the hyperspace density.

4.5 Distance and Overlap Between Clusters

(Compute volume of intersection. If successful, substitute plots. By computing the volumes of the intersections we know the overlap exactly, instead of just knowing "the larger $diff$, the larger the overlap". However, we currently at least know THAT they overlap, and that the real users overlap more than the

Todo:

artificial users. But knowing how much they overlap exactly, and maybe express it as a fraction of something, would give even more accurate insight.)

Section 4.4 gives a first idea of the nature of the MSS. However, the findings do not explain why most songs of the group playlist are liked by all group members, as shown in Section 4.2. In order to find out how close clusters lie together, and how much they overlap, we compute the smallest distances between every pair of clusters, and we also record the radii of those two clusters, which gives us a measure for the overlap. The results in Sections 4.2 and 4.3 suggest that there is a lot of overlap between the music tastes of real users, and substantially less for artificial users.

Figure 4.7 shows how the experiment is performed. For each cluster, we find the closest cluster, i.e., we find all pairs of clusters that each minimize the distance between their centers $D_{i,j}$ (blue line in Figure 4.7). For each of those pairs, we also compute the sum of the radii, i.e., $R_{i,j} = r_1 + r_2$ (red lines in Figure 4.7). If $diff = R_{i,j} - D_{i,j} > 0$, the two clusters overlap. The larger $diff$, the stronger the overlap is. If $diff \leq \|r_i - r_j\|$, the smaller cluster lies completely within the larger one.

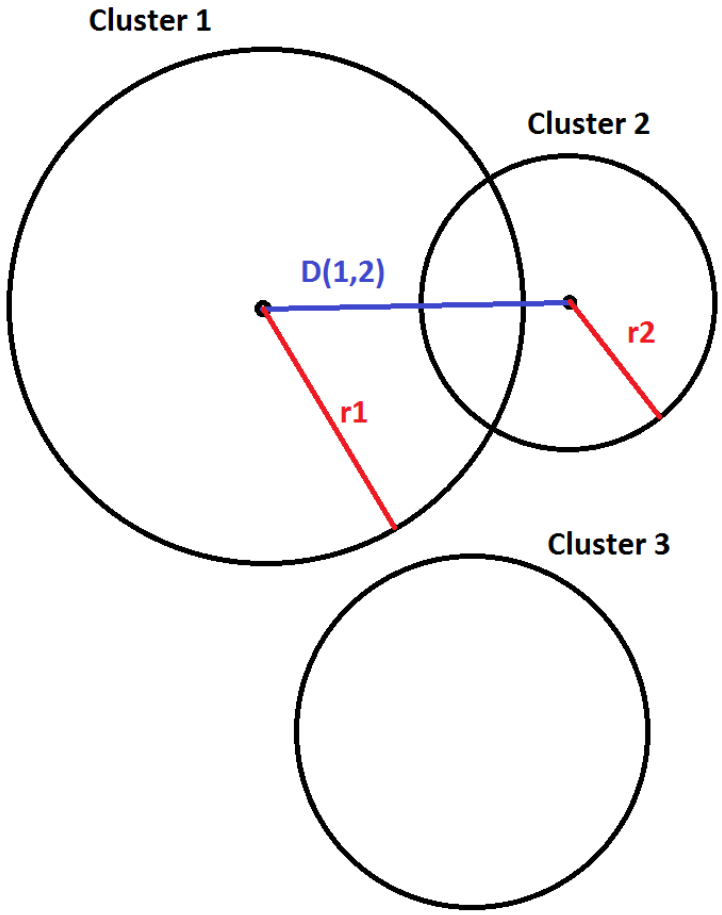

Figure 4.7: Visualization of cluster-overlap experiment.

Section 4.5.1 shows how much the clusters of single users' music tastes overlap, i.e., how densely "populated" the MSS of a single user is. In Section 4.5.2 we performed the experiment again, but this time we compared the clusters of one user to those of another one, which should help us to explain the results of Sections 4.2 and 4.3.

4.5.1 Clusters Within Music Taste of Single Users

In order to find out how the clusters are distributed in the MSS, we looked at the inter-cluster distances and their radii for all users.

For each cluster c_i of user u_1, the distance $D_{i,j}$ to the closest cluster c_j of user u_1 is computed. Additionally, the sum of the two respective cluster's radii ($R_{i,j} = r_i + r_j$) is recorded. If $diff_{i,j} = R_{i,j} - D_{i,j} > 0$, the two clusters c_j and

c_i overlap. The smaller $diff_{i,j}$ is, the greater the overlap.

Figure 4.8 shows $diff_{i,j}$ for all clusters of all real users. The values were sorted to create a clearer plot. The Figure shows that for all 1000 clusters, only a very small fraction does not overlap with any other cluster. For many clusters, the overlap is significant. This leads to the assumption that a user's MSS is very densely populated with clusters. This also means that songs can lie in more than one cluster.

Todo: Define what "significant" means

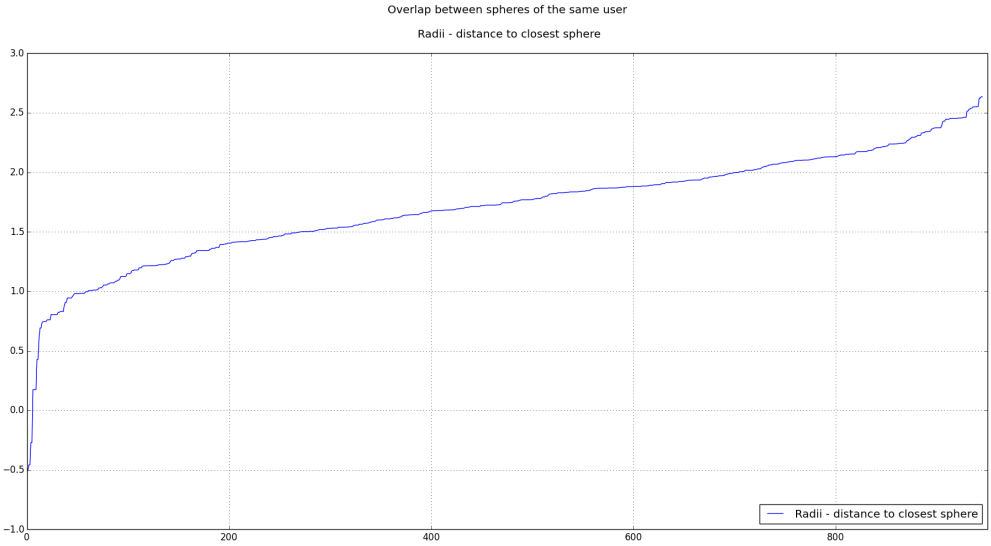

Figure 4.8: Difference between the minimal distance between two spheres and the sum of the respective radii. This gives an idea of the amount of overlap between clusters of one user.

4.5.2 Clusters Across Music Tastes of Two Users

For this experiment, we compare one user's music taste to that of another one, and compute the overlap between the clusters of their music similarity spaces.

For each cluster $c_{1,i}$ of user u_1, the distance D_i to the closest cluster $c_{2,i}$ of user u_2 is computed. Additionally, the sum of the two respective cluster's radii ($R_i = r_1 + r_2$) is computed.

If $diff_i = R_i - D_i > 0$, the two clusters $c_{1,j}$ and $c_{2,i}$ overlap. The smaller $diff_i$ is, the greater the overlap.

Figure 4.9 shows the values for D_i and R_i for a sample of real users. One can see that there is not a single case where there is no overlap between spheres (i.e. $diff_i$ is always larger than zero). In most cases the overlap is significant. Figure

4.10 shows the same data, but with the values sorted in an ascending manner, which makes it even more clear that the average overlap is high.

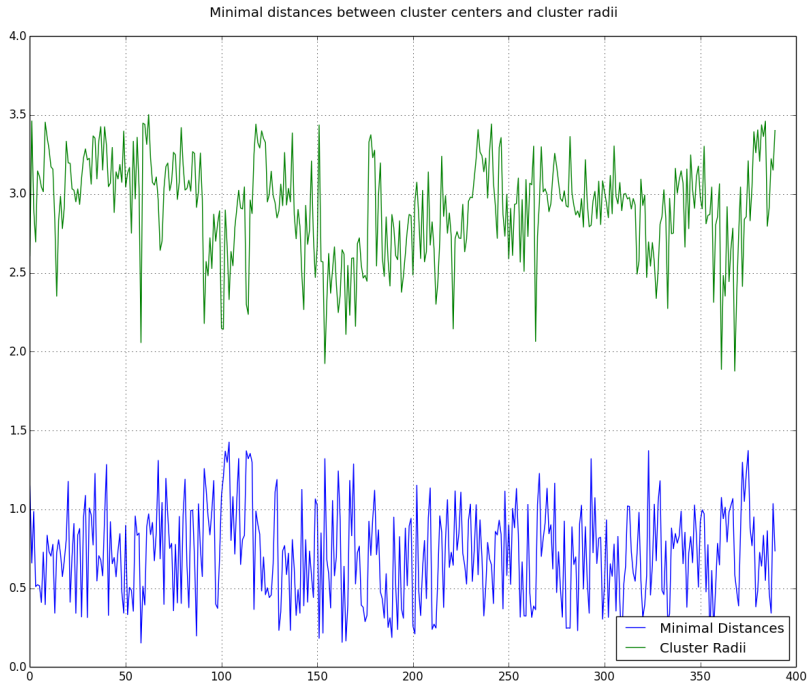

Figure 4.9: Minimal distances between clusters and the sum of the radii for real users.

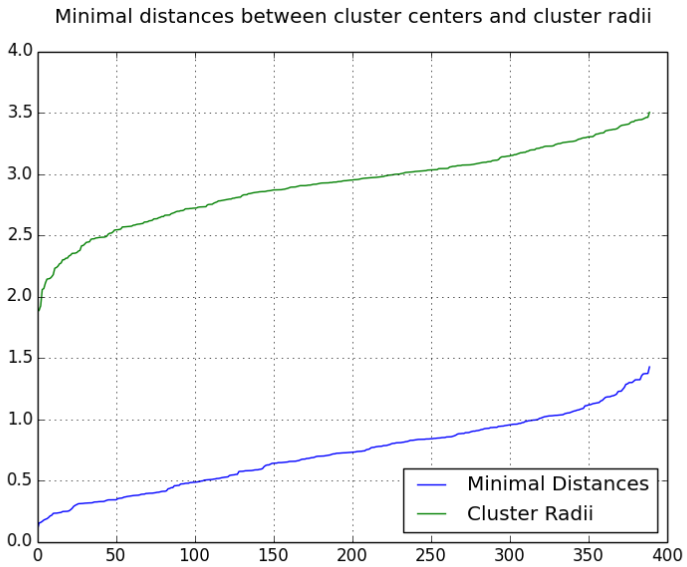

Figure 4.10: Minimal distances between clusters and the sum of the radii for real users, both sorted in an ascending manner to make it more clear.

Figures 4.11 and 4.12 show the respective plots for the experiment with `artificial users`. We can see that, even for users deliberately created such that they should have disjoint music tastes, almost all sampled clusters overlap with one from the other user. However, there is much less overlap than for the experiment with real user data. This is shown in Figure 4.13, where $diff_i$ is plotted for both real and artificial users.

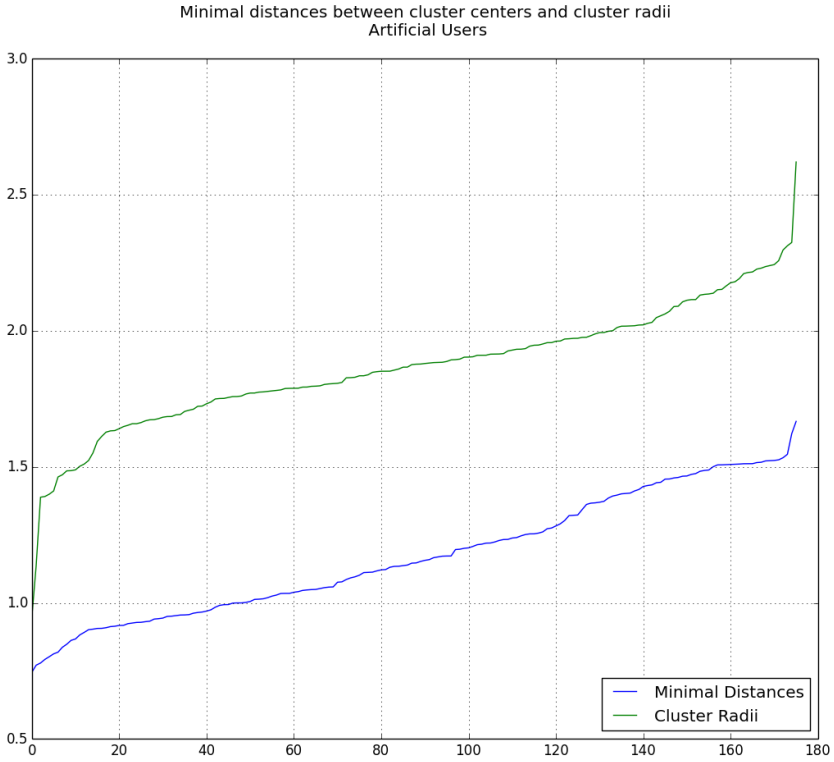

Figure 4.11: Minimal distances between clusters and the sum of the radii between `artificial users`.

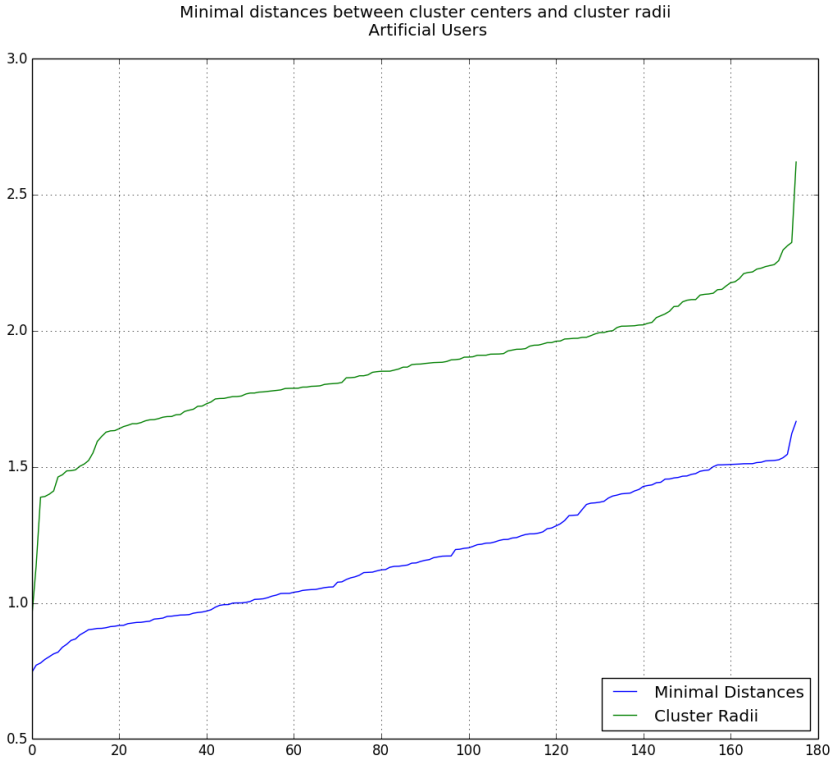

Figure 4.12: Minimal distances between clusters and the sum of the radii between `artificial users`, both sorted in an ascending manner to make it more clear.

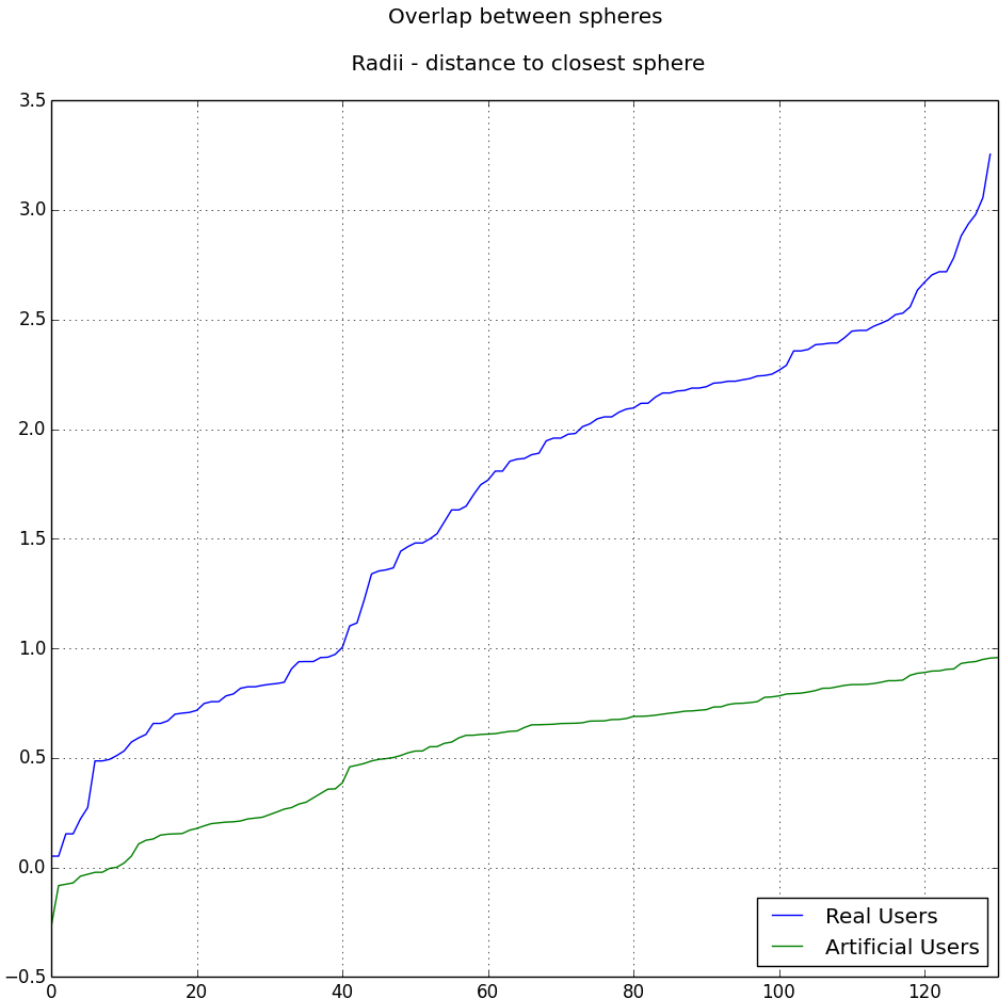

Figure 4.13: Difference between the minimal distance between two spheres and the sum of the respective radii. This gives an idea of the amount of overlap between clusters of different users.

4.6 Analysis

The experiment in Section 4.2 showed that for any group size, a large portion of the songs is liked by all group members. This is in some ways not a good result, because it means that creating a group playlist (i.e. an implicit group music taste) is not much better than picking songs at random. The plots show that roughly 70-85% of the songs are liked by all users, which translates into a 15-30% advantage over randomized song-picking.

A possible explanation is that in a large enough music collection, there will always be some songs belonging to a certain genre, or type of music, especially with common genres like Hip Hop, Pop, electronic music, and so on. If there are three users and every user has some Hip Hop songs, Pop songs and some electronic songs, they will all have a sphere covering those songs, and therefore representing a music type the person likes. One person could have 1000 Pop songs and only 10 Hip Hop and electronic songs each, while for the other person it could be the other way around. The result would be that both persons like all songs in their combined collection. This is the case for any number of users with a certain number of songs.

It needs to be mentioned that while we only incorporated users with at least 100 songs, the average number of songs is way higher than that, which might strongly affect the results of the experiments. Since the music taste computation is based on `all` songs in a music collection, the resulting music taste can only be an accurate representation of a persons actual music taste, if the music collection only contains songs that the user actually likes. However, people often just download entire albums, or even mix albums, where they might only like a small fraction of the songs. Even worse, the songs within an album (mostly for mix albums) could be very scattered in the music similarity space, leading to an even more inaccurate music taste representation.

The second experiment in Section 4.3 served as a sanity check to make sure the group playlist creation is working as intended. The results support the above theory. In a more controlled experiment, where the users are created in such a way that there should hardly be any overlap between music tastes (and absolutely no overlap between song collections), the results look very different. The resulting plots look like mirror versions of the plots for the real-user experiment. This leads to the conclusion that for our application, the usefulness of the current version of the music taste implementation is somewhat limited.

As was also shown in Section 4.2, reducing the acceptance threshold leads to group playlists with a lot fewer songs that are liked by all group members. However, as mentioned before, it is not clear that reducing the threshold makes sense. We believe that it does not, because what actually happens is that **after** the music taste computation (which already removes outliers), one would artificially exclude songs that actually lie within the spheres. By choosing a threshold $t_{acc} = 0.4$, which means artificially reducing the size of the sphere to 40 percent of its radius, one would effectively exclude around $1 - \frac{0.4^2 * \pi}{\pi} = 0.16 = 16\%$ of all songs, and only `accept` songs that are very close to the cluster centers. This by itself is not bad, however, one does not know if the `right` songs would be excluded, or if lowering the acceptance reduces the quality of the playlist. It needs to be further examined if the songs that are closer to the cluster centers are "better" than the ones on the edges. However, as of now, we do not believe that this is the case. It is difficult to assess the superiority of a song on this level

of granularity anyway.

In Section 4.4 we presented an experiment that gives a rough idea about the density of the music similarity space. At first glance, the numbers would suggest that different users' spheres should not overlap that much, because the densities are of very small magnitude. However, as stated, using an enclosing hypersphere as boundary of the similarity space artificially increases the volume, therefore decreasing the density. Furthermore, it is to be expected that the spheres always lie in the same areas, since they correspond with certain genres, at least to some degree. Rock songs are rock songs and therefore the spheres of users that own Rock songs are expected to overlap. To gain further insight, this experiment should be refined by choosing a smaller enclosing hyper-body.

Section 4.5 shows that while the entire MSS is only sparsely populated with clusters, most of them are very close together, and are highly overlapping. The results in Section 4.5.1 support the assumption that the music taste computation is not very accurate, since the clusters highly overlap, despite the fact that there was no upper limit to the number of clusters for the k-means algorithm during the computation. This means that, in theory, there could have been more and smaller clusters, which would have resulted in less overlap.

The comparison between real users yielded that `almost all` sampled clusters had overlap, and in most cases, the overlap was significant. As expected, for the artificial users, there is much less overlap, but it still exists for almost all sampled clusters. However, as seen in the Figures in Sections 4.2 and 4.3, this difference in overlap translates into big differences in the computed group playlist.

In conclusion, one can say that the music taste computation works correctly, but not optimally. In order to improve the performance of the matching process, one would need to improve the underlying music taste computation process. This would go beyond the scope of this thesis, and we will therefore continue to work with what we have.

Chapter 5

Implementation

This chapter describes how the concept of group playlists is incorporated and implemented within the Jukefox framework.

Section 5.1 gives an overview of the application's architecture. Section 5.2 shows how the server communicates with the clients, and explains the different functionalities that we implemented. Section 5.3 shows how the application is used.

5.1 Architecture

The new group music functionality is entirely integrated into the existing Jukefox applications. The client side runs within the android application, and the server side is integrated in the Jukefox PC client.

Figure 5.1: Visualization of the application's architecture. The server could also be connected to the network via WiFi, as long as all devices are within the same network.

Figure 5.1 shows the architecture of the application. Multiple android phones (clients) running the Jukefox application are connected to the same network as a computer running the Jukefox PC client (server). The mobile phones provide the data (music tastes, music library information, song files) and the server manages the computation and music playback.

5.2 Functionality and Communication

The server communicates with the clients over TCP. The system is event-based, which means that incoming messages trigger events at the receiver. The server and clients communicate via messages with pre-defined actions. Based on the type of the message and the associated action, the receiver triggers the next action. All relevant classes can be converted to XML so they can be sent along in these messages. The receiver then parses the message and extracts the objects again. In addition to these messages, the clients can send raw data, e.g. mp3-files.

Figure 5.2 shows the communication protocol of the application. There can be several clients communicating with the server simultaneously. In that case, the server waits for all currently connected clients to send their song coordinates (step 7) before it computes the group playlist. Analogous, the server waits for all mp3-files to arrive before starting playback. When a new person joins the group, steps 1-6 are redone for the newly connected client. Steps 7-9 are done

for all group members again.

The server opens one port to which all the clients connect to. This connection is used to establish a new port on which all subsequent communication is running. There is one such `sharing channel` for each client. This channel stays open, such that the server can request new songs from already connected clients at any time.

In step 7, the server computes the group playlist and decides which songs to request from the connected clients. The server generally just picks songs from the top of the group playlist, which as described in Section 3.3.2 is ordered by the like-count. If, as described in Section 3.3.3, one user's karma gets to high, the server picks a song that said user likes.

For the song requests in step 8, the requests are distributed as evenly between the connected clients as possible, in order to speed up the process. If multiple users own the same song, it is requested from the client that has received the fewest requests so far. Once all requested files have been received, the server starts the playback.

At any point in time, clients can leave the group as in step 11. The server then removes that user from the group, and recomputes the playlist to fit the remaining group members.

The server periodically sends still-alive requests to all connected clients. If a client does not respond, he is removed from the group just as in step 12. The keep-alive requests and the clients' responses are not shown in Figure 5.2. This makes sure that people that left the party are removed from the group after some time, and keeps the playlist up do date.

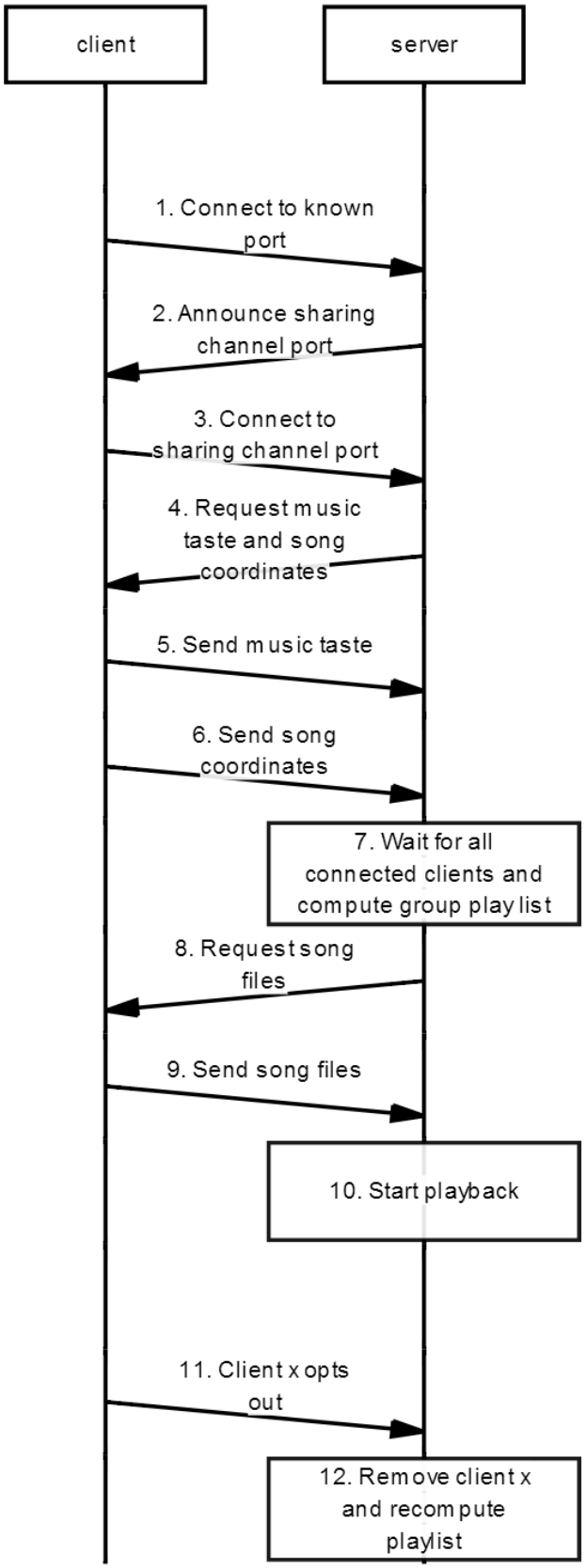

Figure 5.2: Application flow and communication protocol.

5.3 Usage

This section shows how to use the new group-music functionalities of Jukefox:

1. Start the Jukefox PC client on the server
2. Type command `groupMusic` to start a thread that is listening for incoming connections by clients. See Figure 5.3.
3. Users of the Jukefox Android application can now connect to the server as seen in Figure 5.4.
4. The server and clients now exchange messages and information as described in Section 5.2. Since all this runs in a separate thread, the client application can be used normally, e.g. for playing music.
5. Once all mp3-files are received, the server-side application creates a group playlist as described in Section 3.3.2 and converts it to a playlist that can be used by the Jukefox PC client.
6. New users can join any time, which triggers the group playlist computation anew, and once the computation is done, the new group playlist is loaded and starts playing.

Figure 5.3: Jukefox PC client command line interface. The `groupMusic` command starts a new thread that listens for incoming connections.

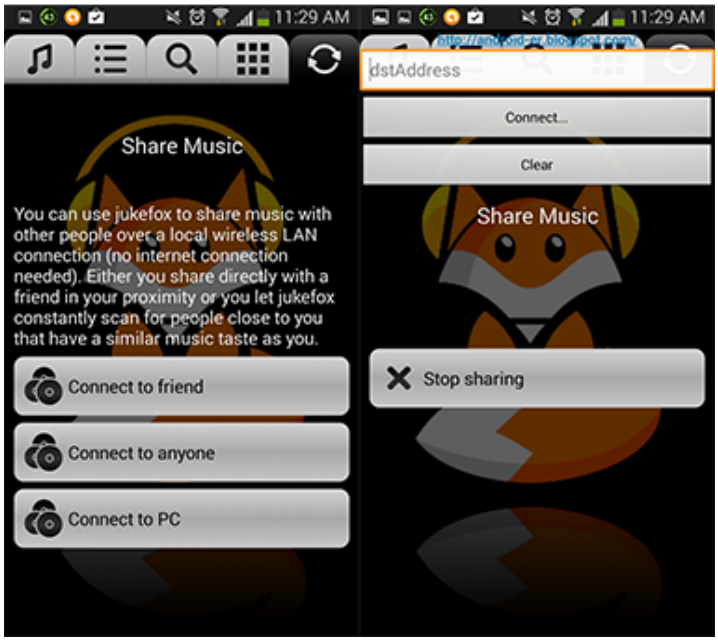

Figure 5.4: Jukefox application screens for the clients to connect to the server.

Chapter 6

Outlook

The presented concepts and ideas for the computation of a group music taste are a good starting point, and the integration within Jukefox works rather well. However, there are possibilities for improvement, and areas where further research is needed. This chapter addresses a few of those.

6.1 Music Taste Computation and Music Similarity Space

As shown in Section 4.2, the use of the Jukefox music taste computation for computing a group playlist, yields a playlist where a large portion of the songs match `all` group members' music tastes. This indicates that the music taste representations are not accurate enough. Section 4.5 showed that there is a lot of overlap between users' music tastes, even for the artificial users, which supports our notion that the music taste computation, i.e., the k-means clustering, could be improved. However, real-world experiments with the group music functionalities are needed to confirm that. It could very well be that real users are satisfied with the resulting group playlists, even though our numbers "only" show a 15-30% improvement over randomized song-picking, as discussed in Section 4.6.

6.2 Group Music Taste

Instead of implicitly computing a group music taste by creating a group playlist, one could think about computing an actual parametrization of a group's music taste. The end result would be the same; a playlist that suits the group. However, a group music taste object could be used, stored and shared just like the music taste objects for single users, which could be an advantage for future applications. So far, the only thing that can be stored for future use, apart from the mp3-files, are the playlists for the Jukefox PC client.

Chapter 7

Conclusion

The newly implemented functionalities allow Jukefox users to get together in groups and listen to music they all enjoy, without having to argue over it. The application requests data from the group members and computes the group playlist, directly requests the best songs and plays them. This all happens automatically, the clients only have to be in the same network as the server, enter an IP address and then press a button. The application periodically checks if the group members are still in the network, and if not, removes them from the group. This ensures that the playlist stays up do date over the course of hours independent of people coming and going.

We also evaluated the performance of the music taste computation itself, and found that our application could benefit from improvements. As of now, we only disregard 15-30% of the songs, while the rest is assumed to be liked by all group members. From personal experience, this does not reflect the real-world situation. People are usually more picky about music, and do not like around 80% of `all` songs there are.

Chapter 8

Presentation Slides

Bibliography

[1] Dunning, A.: Beethoven or Britney ? - A sociological exploration of music taste, cultural consumption and social class. PhD thesis, The University of Manchester

[2] Mulder, J., Bogt, T., Raaijmakers, Q., Vollebergh, W.: Music taste groups and problem behavior. Journal of Youth and Adolescence **36**(3) (2007) 313–324

[3] Peeters, G.: A large set of audio features for sound description (similarity and classification) in the cuidado project. Technical report, Ircam (apr 2004)

[4] Foote, J.T.: Content-based retrieval of music and audio. In: MULTIMEDIA STORAGE AND ARCHIVING SYSTEMS II, PROC. OF SPIE. (1997) 138–147

[5] Hoashi, K., Zeitler, E., Inoue, N.: Implementation of relevance feedback for content-based music retrieval based on user prefences. In: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval, ACM (2002) 385–386

[6] Hoashi, K., Matsumoto, K., Inoue, N.: Personalization of user profiles for content-based music retrieval based on relevance feedback. In: Proceedings of the eleventh ACM international conference on Multimedia, ACM (2003) 110–119

[7] Hoashi, K., Matsumoto, K., Sugaya, F., Ishizaki, H., Katto, J.: Feature space modification for content-based music retrieval based on user preferences. In: Acoustics, Speech and Signal Processing, 2006. ICASSP 2006 Proceedings. 2006 IEEE International Conference on. Volume 5., IEEE (2006) V–V

[8] Hoashi, K., Ishizaki, H., Matsumoto, K., Sugaya, F.: Content-based music retrieval using query integration for users with diverse preferences. In: ISMIR. (2007) 463–466

[9] Logan, B., Salomon, A.: Music similarity function based on signal analysis (dec 2002) US Patent App. 10/004,157.

[10] Logan, B.: Music recommendation from song sets. In: ISMIR. (2004)

[11] Grimaldi, M., Cunningham, P.: Experimenting with music taste prediction by user profiling. In: Proceedings of the 6th ACM SIGMM International Workshop on Multimedia Information Retrieval. MIR '04, New York, NY, USA, ACM (2004) 173–180

[12] Kuhn, M., Wattenhofer, R., Welten, S.: Social audio features for advanced music retrieval interfaces. In: Proceedings of the International Conference on Multimedia. MM '10, New York, NY, USA, ACM (2010) 411–420

[13] von Bergen, P.: On the feasibility of opportunistic ad hoc music sharing. Master's thesis, ETH ZÃ$\frac{1}{4}$rich (aug 2012)

Appendix A

Appendix Chapter