



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*Distributed
Computing*



Content Sharing using Cooperative Playlists and Social Quizz Games

Bachelor's Thesis

Andreas Hess

`ahess@student.ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:

Pascal Bissig, Laura Peer
Prof. Dr. Roger Wattenhofer

September 3, 2015

Acknowledgements

Firstly I sincerely thank my two supervisors Pascal Bissig and Laura Peer for their great support, guidance and motivation through the whole project.

Besides my supervisors, I would like to thank Prof. Dr. Roger Wattenhofer for his valuable insight and help during the evaluation phase.

I express my sincere gratitude to all members of the Distributed Computing Group at ETH Zürich who helped testing and evaluating the project's outcome.

Last but not least I thank my parents, who support me in every live situation, and my partner Rahel for her never ending helpfulness and her understanding and reassurance in stressful situations.

Abstract

We tackle the task of building a platform that allows users to easily find and share interesting video content. To do so we introduce two tools: cooperative playlists and a social quizz game. We present a novel design of how to set the logic behind cooperative playlists next to the design of a completely new game. For both the cooperative playlists and the game we describe the implementation as web application.

Contents

Acknowledgements	i
Abstract	ii
1 Introduction	1
2 Cooperative Playlists	3
2.1 Source of Titles	3
2.2 Dynamic Playlists	4
2.2.1 Capturing Popularity	5
2.2.2 Calculating the Rank	6
2.3 Listening Modes	7
2.3.1 Radio	7
2.3.2 Dictator	8
2.4 Public vs. Private Playlists	8
3 Game	9
3.1 Basic Mechanics	9
3.1.1 States and Transitions	11
3.1.2 Picking a Video	14
3.2 Incentives	15
3.2.1 Voting Points	15
3.2.2 Confusion Points	16
3.2.3 Missing Title Penalty	17
3.3 Game Speed	17
3.4 Information Sharing	18
3.5 Bots	18
3.5.1 Random Bot	19

3.5.2	Copypcat Bot	19
3.6	Observe Mode	19
3.7	Rating Videos	20
4	Application Design	21
4.1	Libraries Used	21
4.1.1	Server Side	21
4.1.2	Client Side	22
4.2	Connectivity	22
4.3	Authentication	22
4.4	Views	23
4.4.1	Lobby	23
4.4.2	Playlist	25
4.4.3	Game General	29
4.4.4	Game State Specific - <i>Voting Phase</i>	30
4.4.5	Game State Specific - <i>Scores Phase</i>	32
5	Prototype Iteration	34
	Bibliography	36

Introduction

The mass of songs and videos in the internet is enormous and without the help of specialised tools or platforms, an individual stands no chance in grasping the personally relevant and interesting part of that content.

User based recommendation systems like reddit or Imgur are very popular. Reddit had over 200 million unique visitors last month [1], Imgur hit the 100 million unique visitors per month mark in 2013 [2]. We will apply the principle of user based recommendation to songs and videos using two kinds of tools: cooperative playlists and a social quizz game.

Besides traditional playlists that are created and controlled by a single person, there are other types of playlists like cooperative playlists or machine generated playlists. Cooperative playlists aim towards the involvement of more than one person for controlling its content. The content of machine generated playlists is added by some algorithm that tries to capture a user's taste and find similar content. There are several existing platforms for such playlists:

Spotify: Spotify offers "collaborative playlists" [3]. When creating a playlist, the user can choose to make it a collaborative playlist. If he does, this playlist can then be shared with other users which are allowed to change the playlist, i.e. to add or remove songs. This approach only works in a small and private environment. Also the songs are all treated the same, which means all songs are played with the same probability and a user's rating for a song has no influence on that probability.

Pandora Radio: Pandora is a personalised internet radio that tries to find similar songs to an initial song suggested by the user [4]. The user can then rate those played songs which helps Pandora at capturing the user's taste and to improve the matching when looking for similar songs. Pandora's algorithm for finding similar songs is based on the Music Genome Project [5] where songs are analysed using up to 450 distinct musical characteristics. In this work we focus on user created recommendations rather than content based recommendations.

Plug DJ: Plug DJ is a platform based on Youtube videos. Users can join (user created) rooms where they take turn at playing the role of the DJ [6]. The DJ changes after each played video. In order to avoid inappropriate or unwanted videos from being played, Plug DJ introduced the "meh" algorithm [7] that allows listeners in a room to skip those videos more easily (amongst other improvements). In this work we will aim towards cooperative playlists which base the choice of the played title on user ratings instead of picking a user and letting him choose a title. Also we will describe a special mode of playlists where an individual user has more control over what is being played.

Like for cooperative playlists, there are several platforms for quizz games but none of those seem to aim towards sharing interesting music or video based content. We will describe an approach of a quizz game that is very similar to a cooperative playlist and in which users have to use their knowledge about co-players in order to win.

For both the cooperative playlists and the quizz game we created a web application in order to test and evaluate our design and decisions.

Cooperative Playlists

Playlists act as a container for playable content. Basic functions of a playlist are to allow adding, removing and finally to play the content in some order. Personal playlists are ruled by a single person. This person is the only one to manipulate the playlist or to play it. This concept can be relaxed by allowing a group of people to listen to the playlist while there is still only one person allowed to manipulate it. Cooperative playlists are about opening the concept of playlists to groups of people, where each person is allowed to both manipulate the playlist and to play it. In this chapter we will describe a possible implementation of cooperative playlists based on an web application.

2.1 Source of Titles

We considered several platforms as possible source of videos or songs, the main three of which were Youtube, Spotify and Rdio. There were several aspects we had to take into account when making our decision. Firstly, how well developed is the API of the platform and does it allow the needed operations? Secondly, how much content does each platform provide? Thirdly, do we want to use videos or songs? And forthly, is the service free or is there a monthly fee to be payed by the user? The last point is of importance because users would probably be less motivated to try our platform if there was a fee to be payed. As most people will not already own such a subscription, a monthly fee would diminish the potential user base of our platform.

Rdio: Rdio is a music platform where full access costs a monthly fee of 12.95 CHF [8]. The service offers more than 35 million songs and serves 85 countries [9]. There are no official numbers on the subscriber base. The API seems sufficiently elaborated but the developer community seemed to be rather small.

Spotify: Spotify is very popular in Europe and the U.S. As Rdio does, it offers a premium subscription plan for 12.95 CHF per month [10]. There

are more than 20 million paying subscribers and over 30 million available songs [11]. The main problem is the rather restrictive API. Spotify does not allow to embed a configurable player into a website but only the so called Spotify Play Button. This button does not allow to dynamically change the content that is being played which crosses Spotify off the candidate list.

Youtube: Youtube is free, so there is a huge potential user base. The other main difference to the other two platforms is that Youtube provides videos instead of songs. But as most of the songs are available on Youtube as a video this can be considered as an extension to just providing songs. Videos would bring additional potential like the possibility to create playlists containing funny content. The Youtube API offers a configurable IFrame Player API [12] and also a very rich data API [13]. However there are videos on Youtube that cannot be embedded into external sites.

After these considerations, we decided to use Youtube as our source of videos. Its API provides all the functionality we need and as it is free to use, the potential user base would be very large.

2.2 Dynamic Playlists

In a cooperative playlist, there will expectedly be good and bad content. In this context good means liked by most users, bad means disliked by most users. Our playlists should adapt the order of videos according to each video's popularity in a way that popular videos are more likely to be played. In the following section we describe an attempt to achieve this using user ratings to estimate the popularity of a video.

We introduce a novel method how to generate dynamic playlists. Users can join playlists and watch the content there, they can add and rate videos. There are two possible ratings, either positive or negative. Each video carries several attributes, two of which are `pure_rating` and `last_played`.

`pure_rating` is calculated from the ratings of users and therefore represents the popularity of the video. `last_played` is an abstract representation of the amount of time that passed since the video was last played. It is a counter that holds the number of videos that were played since the particular video was played last. The value is needed because we do not want a playlist that always plays the most popular video if there are other videos available. If the counter's value is large, the owning video is more likely to be played again.

The mentioned attributes are needed to compute the rank of a video. The video with rank one is at the top of the playlist and will be played next, the

video with rank n (where n is the number of videos in the playlist) is at the bottom of the playlist.

2.2.1 Capturing Popularity

An essential question is how to compute the popularity when provided with the number of positive and negative user ratings. A video with 7 positive and 3 negative ratings should have a higher popularity than a video with 3 positive and 7 negative ratings. But what about a video with 14 positive and 6 negative ratings compared to a video with 7 positive and 3 negative ratings? We assume that each video has a true probability of being liked by a player. We call this value *acceptance*. The *acceptance* of a video can also be measured empirically as shown in Equation 2.1.

$$acceptance = \frac{\#\{positive_ratings\}}{\#\{positive_ratings\} + \#\{negative_ratings\}} \quad (2.1)$$

The measured *acceptance* is exactly the same for the first and the second video that were mentioned in the question above. But according to the law of large numbers, the measured *acceptance* of the first video is probably more close to the real *acceptance* of that video. That is why we put more trust into the measured *acceptance* of the first video and therefore give it a higher popularity.

We use a slightly modified version of an algorithm that computes the lower bound of the Wilson score confidence interval for a Bernoulli parameter to compute *pure_rating* of a video [14]. The algorithm answers the question "Given the ratings I have, there is a 95% chance that the "real" fraction of positive ratings is at least x ". For completeness, the formula is listed in Equation 2.2.

$$x = \left(\hat{p} + \frac{z_{\alpha/2}^2}{2n} - z_{\alpha/2} \sqrt{[\hat{p}(1 - \hat{p}) + z_{\alpha/2}^2/4n]/n} \right) / (1 + z_{\alpha/2}^2/n) \quad (2.2)$$

γ stands for the calculated popularity, \hat{p} denotes the observed fraction of positive votes and $z_{\alpha/2}$ is the $(1 - \alpha/2)$ quantile of the standard normal distribution. For our implementation we chose a 95% confidence interval which means $\alpha = 0.05$ and consequently $z_{\alpha/2} = 1.96$.

Figure 2.1 shows a heat map of the lower bound of the Wilson score confidence interval as a function of positive and negative ratings. In order to use the described algorithm to compute the popularity of a video, it has to be slightly modified. The unmodified version yields popularity 0 whenever a video has 0 positive ratings and x negative ratings where $x \in \mathbb{N}$. This is not the behaviour we wish as a video with 0 positive and 10 negative ratings should have a higher popularity than one with 0 positive and 20 negative ratings. Adding the integer 1 to the number of positive ratings whenever the popularity is calculated fixes this issue at the cost of a slightly modified popularity function. It is equivalent

to considering the scale on the right side of the heat map in Figure 2.1 instead of the one on the left side.

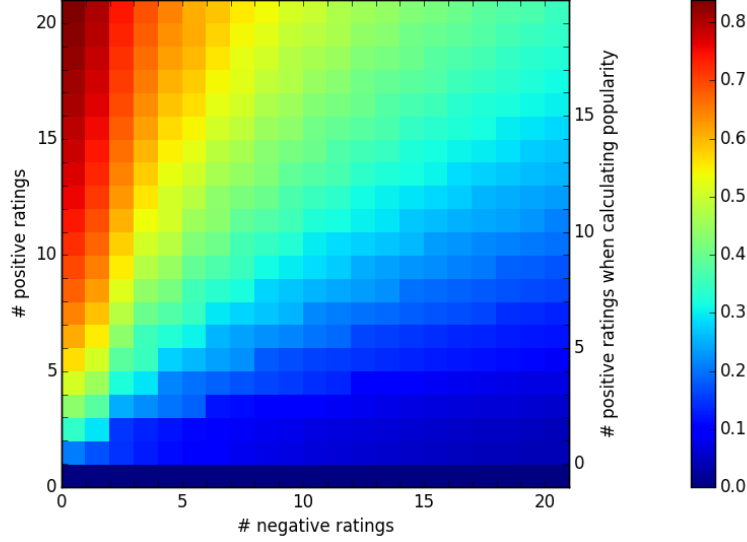


Figure 2.1: A heat map showing the lower bound of the Wilson score confidence interval as a function of positive and negative ratings for the left hand side ordinate scale and the video popularity for the right hand side ordinate scale.

2.2.2 Calculating the Rank

As mentioned earlier it is not enough to calculate the popularity of a video as we do not want the most popular video to be played again and again. Therefore, videos have to be somehow penalised after being played so they will not be played again immediately. Our method achieves this by modifying the calculated popularity which results in a final score for a video. This score will then be used to calculate the rank of the video (which is just sorting the videos by score descending and then numbering them in that order from 1 to n).

In order to penalise recently played videos, we multiply the popularity with a negated inverse exponential function which takes `last_played` as argument and recovers to the value 1 as `last_played` grows to ∞ . The exact formula of the score is given in Equation 2.3. The constants are just an initial guess without experimental background and would have to be evaluated and improved by collecting feedback from users. A plot of the penalty factor function is shown in Figure 2.2.

$$score = popularity * (1 - 0.7 * e^{-0.2 * last_played}) \quad (2.3)$$

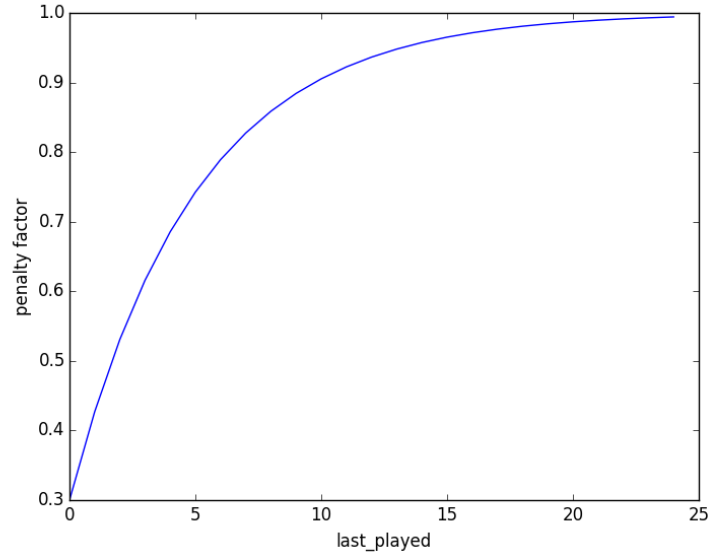


Figure 2.2: A plot showing the penalty factor function which is used to determine the score of a video.

Currently, the penalty factor is independent of the number of videos in a playlist. It might make sense to adjust the formula in a way that videos in large playlists have a lower score after being played than in small playlists. But just like the constants, this question would be subject to user feedback and testing.

2.3 Listening Modes

2.3.1 Radio

This listening mode is inspired by classic internet radio. All users of a playlist see the same video at the same time, i.e. all users are synchronised. The radio mode is very passive, all the user can do is to rate videos and add new ones. He can not simply play the next video if he does not like the current one. But as no possibility to skip at all would be very restrictive, for example in case a 10 hour Nyan Cat video is chosen to be played, there is a consensus based skipping. If all users agree on skipping the current video then this wish will be granted and the next video in ascending rank order will be played.

2.3.2 Dictator

In dictator mode the user is decoupled from other users. There is no synchronisation and if the user wants to skip a video he can do so without restrictions. Additionally, the rank of videos is calculated in a different way. Instead of sorting by score descending, videos are first sorted by the user's rating and then by score. That way, the user will only see videos which he did not downrate unless he downrates every single video in the playlist.

2.4 Public vs. Private Playlists

A key question of a cooperative playlist is who will be allowed to join the playlist and add or rate videos. A playlist could be used by a group of people to exchange content or to watch the same video at the same time while not being at the same place. This group might want to have a private room to do so where no other users are allowed. For example consider a group of friends that creates a playlist called "Friday night hangout" and uses it as a platform to share content for a hangout. However, as all content is added by users and there is no "machine" searching the web for suitable videos, it makes sense to have an open system that allows all interested users to contribute to a playlist. Such a public playlist could for example serve as a container for music of a specific genre like "Scottish Pirate Metal".

To serve both of those purposes we created public and private playlists. Public playlists are open to anyone. Private playlists can only be joined by providing a passphrase (which is chosen when creating the playlist) to ensure that no evil forces can enter. So far private playlists are meant purely for a group of people who have some way to communicate as there is no built in possibility to share the passphrase.

Game

Cooperative playlists are a promising tool to find new and interesting content. But there might be users who want to do this in a more interactive and thrilling way. So besides a new approach to implement cooperative playlists, we also present a social quizz game based on Youtube videos. We integrate cooperative playlists and the game into a single application. The game is meant to be played amongst a group of friends as knowledge about the "video taste" of other participants is of essence for a player. That is why we included the entry point to the game into private playlists, as private playlists are meant to be used by a group of acquainted people. Each private playlist has exactly one game instance. Figure 3.1 shows a screenshot of a private playlist. All of its elements will be explained in detail in Section 4.4.2. The game can be joined using the game widget (we will call it game tab). As soon as the game starts, the user view changes to the *game view*. When the game ends, the user view switches back to the view in Figure 3.1.

3.1 Basic Mechanics

The basic game flow is as follows: each player has to put a video of his choice into a video queue. The game server then picks one of those videos and shows it to all players. Each player has to guess who the uploader of that particular video might be among the current players. The objective of a player is to collect as many points as possible. Both the uploader of a picked video and other players can earn points, based on different criteria. The uploader gets most points if he chooses an ambiguous video that leads half of the other players to identify him correctly as the uploader. Other players get points if their vote is correct. Those criteria will be explained much more in detail in Section 3.2.

The game is partitioned into game rounds. A game round consists of the following actions: picking a video to play, playing the video and collecting votes and finally calculating and displaying each player's vote and earned points.

As mentioned earlier, private playlists have a *game tab*. The *game tab* allows

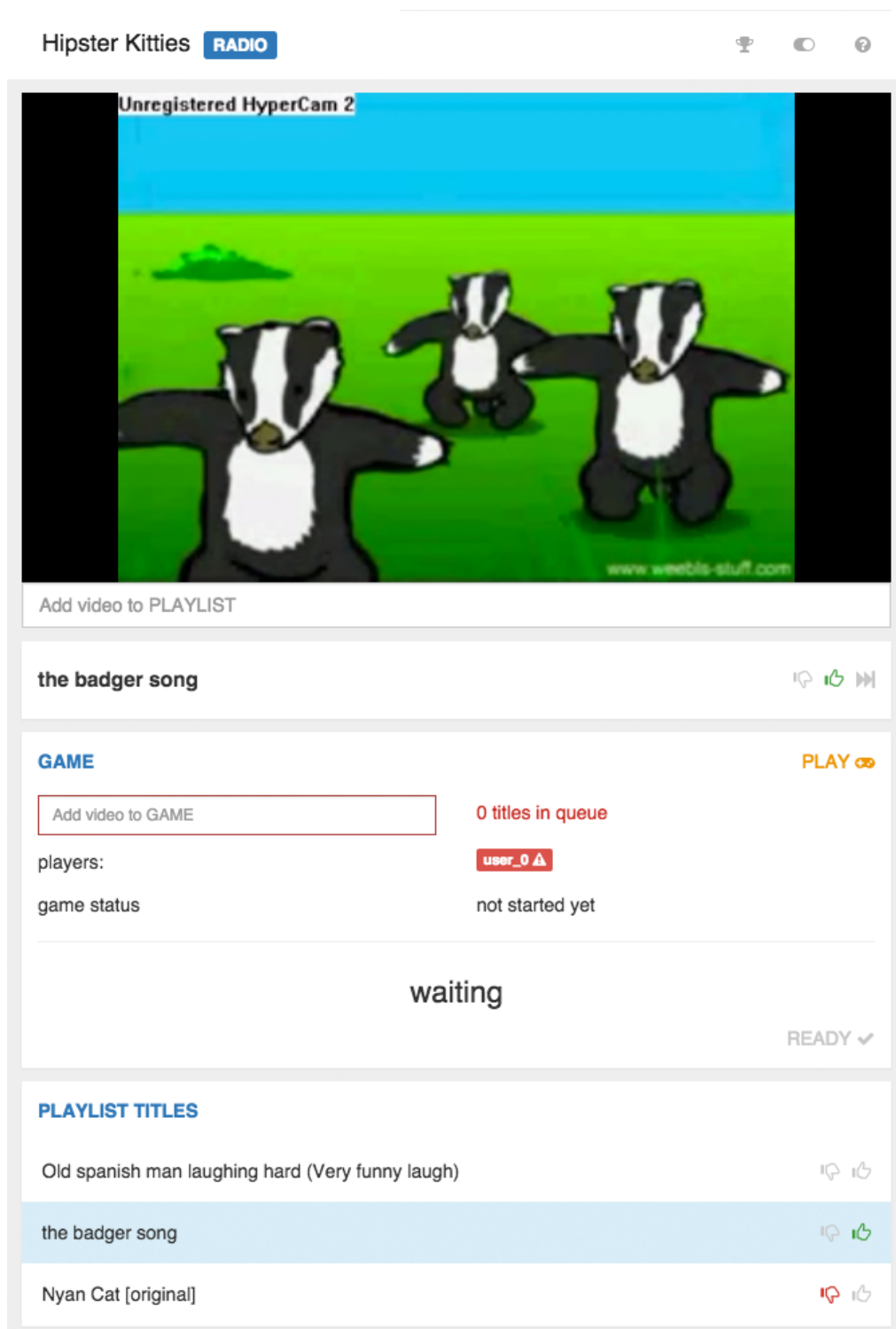


Figure 3.1: A screenshot of a private playlist with an open game tab. The displayed video is taken from Youtube [15].

the user to participate in the game, leave the game and to upload videos. There are three important buttons placed in the game tab, namely the *play*, the *ready* and the *leave* button. The *play* button signals an interest in participating in the game. After clicking it, the user sees other players who are interested in playing and he can add videos to his video queue. Each player has his own video queue. He can add to but not delete videos from it. A video can only be in one queue at a certain time (per playlist). Clicking the *ready* button indicates that the user is now ready to play.

3.1.1 States and Transitions

Both the game and the players can be considered as state machines. There are events that cause them to change state and to listen to a new set of events each of which might again change the state.

Player States:

Figure 3.2 shows the state diagram of a player. The four most interesting states are *Idle*, *Interested*, *Ready* and *In Game*. To join the game, a player needs to be *Ready*, which means to have clicked *ready* and to have at least one video in his video queue.

In Game is divided into three substates. The player starts in *In Game Vote Not Made*. As soon as he makes his vote, he moves to *In Game Vote Made*. A vote is final, this means it can not be changed and the player can't move back to *In Game Vote Not Made*. When no more votes can be made, the player moves to *In Game Scores Phase*.

Game States:

The game can be found in three different states: *Idle*, *Voting Phase* and *Scores Phase*. Figure 3.3 shows a state diagram of the game including events that cause a change of state. *Idle* represents the state when the game does not actually exist, i.e. when there are no players playing. In *Voting Phase*, a video (chosen from the participants' video queues by the game server) is played and users can vote for who they think the uploader is. The purpose of *Scores Phase* is to display the points achieved by each player during the previous *Voting Phase*.

Both in *Voting Phase* and *Scores Phase* there are timers. When *Voting Phase* starts two timers are started, one (we will call it *voting timer*) from 60 seconds, which is the maximum duration of *Voting Phase*, and one from s seconds where s is the duration of the chosen video (we will call this one the *video timer*). *Voting Phase* ends as soon as either all players made their vote or the *voting timer* has finished. To increase the speed of the game, the *voting timer* is set to the minimum of the current value and 20 seconds as soon as the first two votes

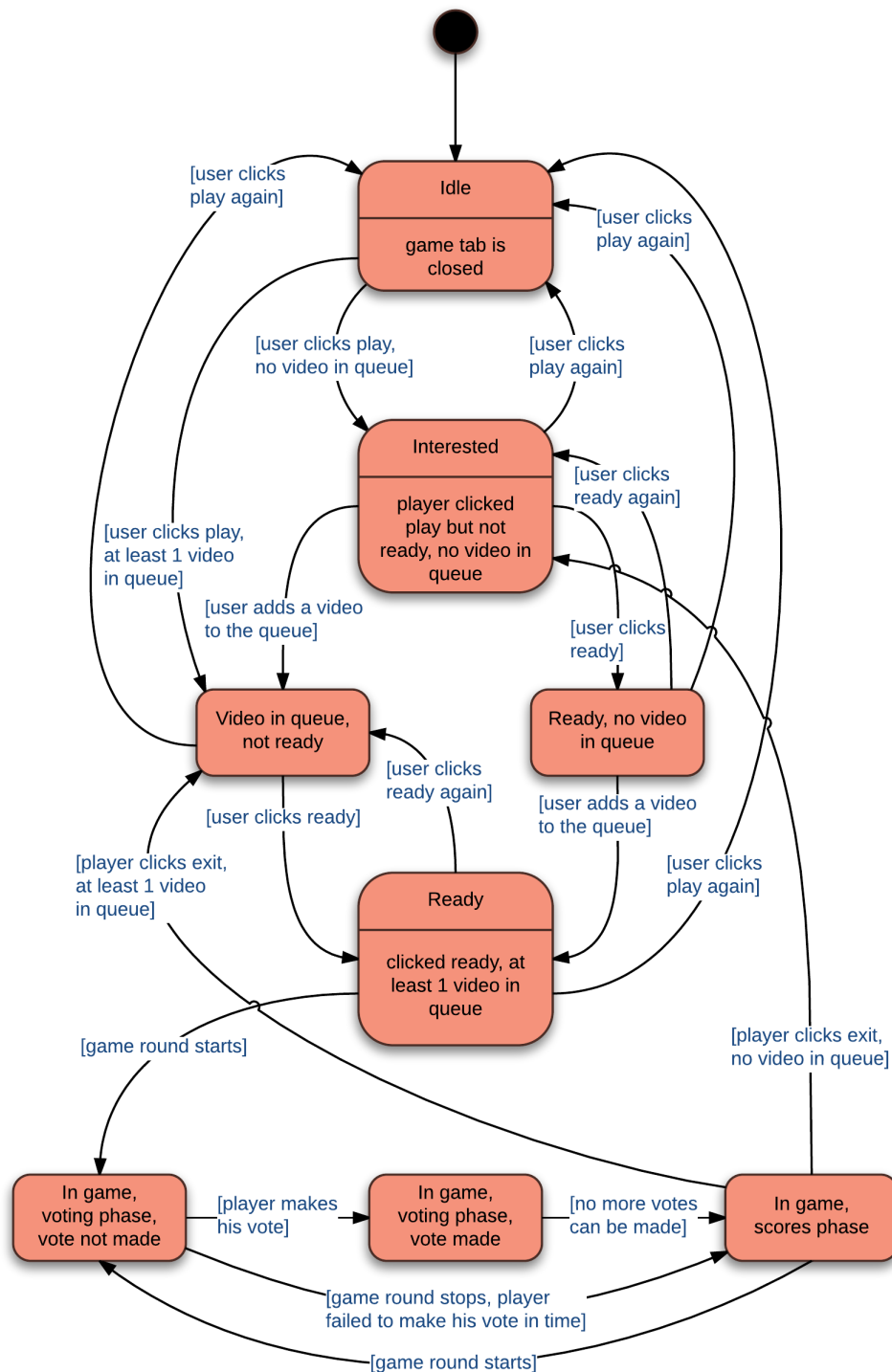


Figure 3.2: A state diagram showing the different player states.

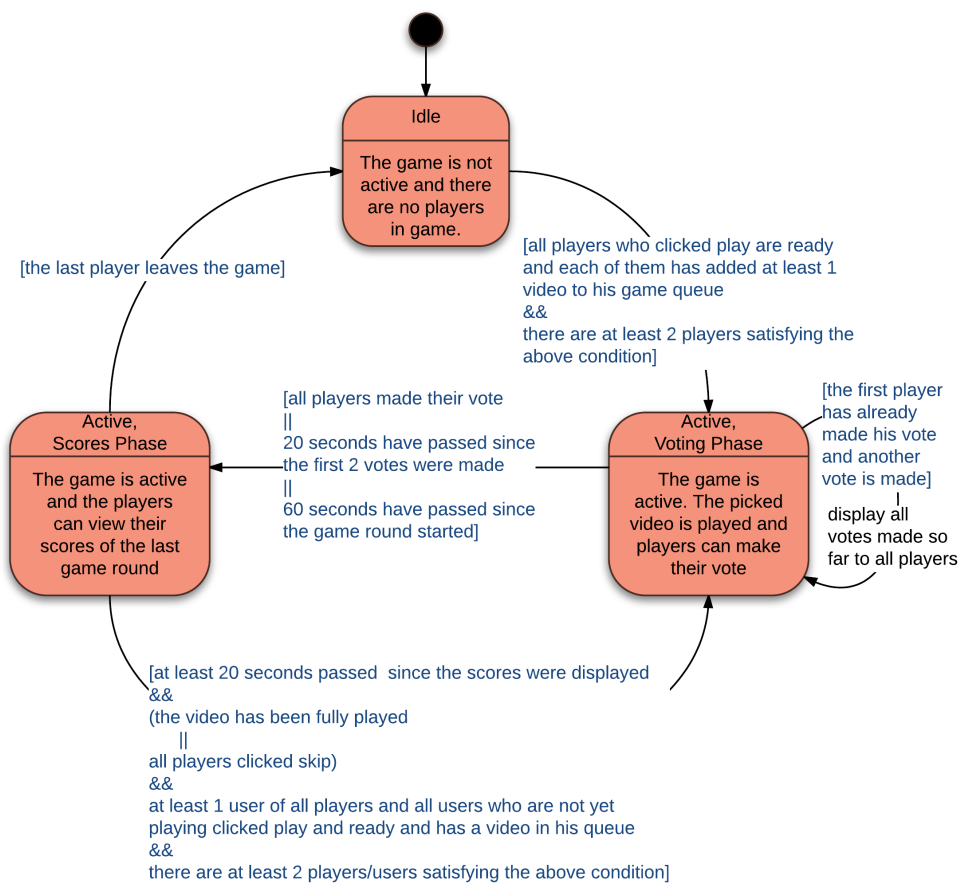


Figure 3.3: A state diagram showing the different game states.

have been made.

Upon entering *Scores Phase*, a timer (we will call it *scores timer*) is started from 20 seconds which is the minimum duration of *Scores Phase*. By default, *Scores Phase* ends as soon as both the *video timer* and the *scores timer* have finished. But as a video might take several hours to be fully played or might just be boring, there is a shortcut. Each player can choose to *skip* this video. If all players do so, then *Scores Phase* can end as soon as the *scores timer* has finished.

As shown in the game state diagram in Figure 3.3, besides the timing there are other conditions to be satisfied when moving to *Voting Phase*. Consider the case in which the game is in *Idle*. In order to move to *Voting Phase*, there have to be at least two players who clicked both *play* and *ready* and have each at least one video in their respective video queue. But because this condition alone would not allow more than two players to play, there is the additional condition that all players who clicked *play* must have clicked *ready* and have put at least one video into their video queue. In other words, the game waits for players who have already clicked *play*.

When starting at *Scores Phase*, the conditions are not the same. To keep the game fluent, we only require one player to have a video in his game queue, although having an empty video queue will be penalised as described in Section 3.2.3. Also, players can join the game between two game rounds, i.e. when moving from *Scores Phase* to *Voting Phase*, players who are not yet playing can join the current game if they are *ready* and have at least one video in their video queue.

3.1.2 Picking a Video

As mentioned previously, the game server picks a video from the players' video queues at the beginning of *Voting Phase*. More precisely, the game server picks a player amongst all participants and then pops the least recently added video from his video queue. When considering the distribution of the players picked, we would want this distribution to be more or less uniform, i.e. each uploader should be picked at more or less the same frequency. But still, the decision should not be predictable as the uploader of a video should not be known to other players.

We try to accomplish a uniform distribution over players by using a heuristics with both its advantages and disadvantages. We keep record of which players were picked in the last 20 game rounds. We call the number of videos picked from player x vid_count_x . For each player, we calculate $\alpha_x = (vid_count_x + 1)^{-1}$. Finally we compute the probability $pick_prob_x$ for a player x to be picked as shown in Equation 3.1 where P denotes the set of participating players. When

deciding on a player, we can simply pick a random number between 0 and 1, align the player probabilities one next to the other in the interval $[0, 1]$ and check whose probability interval is hit by the random number. Then we decide on that player.

$$pick_prob_x = \frac{\alpha_x}{\sum_{p \in P} \alpha_p} \quad (3.1)$$

This approach is easy to implement and only requires us to keep track of the picks of the last 20 rounds. The downside is that players who join a running game have a higher chance of being picked than players who were already playing at that time.

3.2 Incentives

The primary goal for a player is to collect as many points as possible. Points are awarded in each game round at the end of the *Voting Phase*. Players can earn and lose points on different occasions like voting for a player or being the player whose video was picked. There are three different kinds of points: voting points, confusion points and missing title penalty. Each of those kinds will be explained separately in the following sections.

3.2.1 Voting Points

Voting points are awarded for correct votes. If a player correctly identifies the player who uploaded the video he gets voting points. The player who uploaded the video can not earn voting points, i.e. he always gets 0 voting points. The amount of voting points earned depends on the speed of the vote. The first player to vote correctly gets most points, the second gets seconds most etc. The decay is linear with a factor of one, i.e. each correct vote earns one point less than the previous correct vote. The vote of the uploader has no influence on the points earned by the following correct votes. Players who fail to make their vote in time get -2 voting points. These measures ought to motivate players to make their votes at all and to make them quickly.

As the difficulty of identifying the uploader increases with a larger number of players, more points can be earned in games with more players. In fact, the maximum amount of voting points that can be earned by a player is equal to the number of players minus 1. That way, if all players vote correctly, the first vote earns $n - 1$ points (where n is the number of players) and the last vote earns 1 point as the vote of the uploader is not taken into account.

Consider an example with a total of 6 players playing, namely players A , B , C , D , E and F . Assume they give their votes in alphabetic order, that B is the uploader of the picked video and F fails to make his vote in time. The

resulting voting points for each player are illustrated in Figure 3.4. $X \rightarrow Y$ means that player X votes Y . The votes are ordered by time on the abscissa, i.e. the first vote was made by A . B 's vote is correct, however he gets 0 points as he is the uploader of the picked video. A , D and E vote correctly and earn the corresponding amount of points. C makes an incorrect vote and gets 0 points. F fails to make a vote in time and gets -2 points.

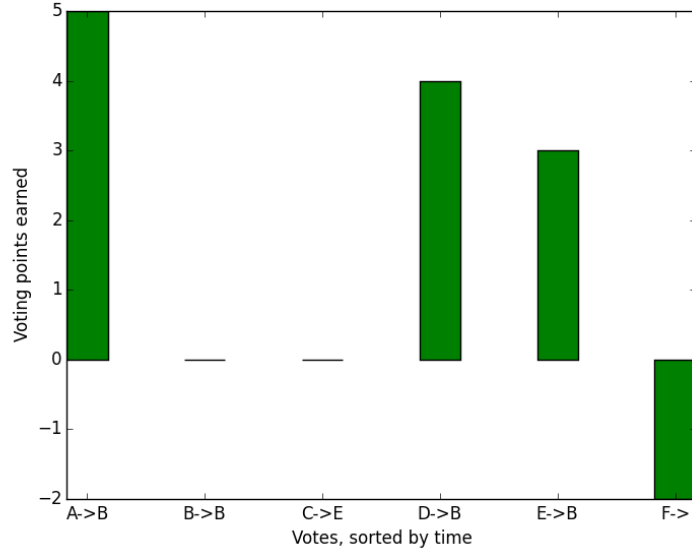


Figure 3.4: Earned voting points for each player action listed on the abscissa for the example scenario in Section 3.2.1.

3.2.2 Confusion Points

With only voting points being there, a good strategy when choosing a video to upload is to select a random video. That way, a player could diminish the chances of being identified as the uploader without even having to spend a thought about choosing a deceptive video. In this context, a deceptive video is one that looks as if it had been uploaded by another player. Uploading deceptive videos might still be a better strategy, but its gain when compared against choosing a random video might not be encouraging enough for the hard work of finding deceptive videos. Therefore, we introduce confusion points.

Confusion points are only given to the picked uploader of a game round. There is a maximum of $n - 1$ confusion points to be earned in each game round (where n is the number of players). In order to get the full amount of points, the uploader has to confuse the other players in a way that exactly half of the players identify him as the uploader. He gets 0 points if either all players identify

him or none of the players do. In between, the points scale is linear. Figure 3.5 shows a plot of the confusion points function. Using this method, the expected reward for uploading random videos or very easy videos (where all players are able to identify the uploader) is very low. This is exactly what we want since random videos and easy videos take the fun respectively challenge from the task of identifying the uploader.

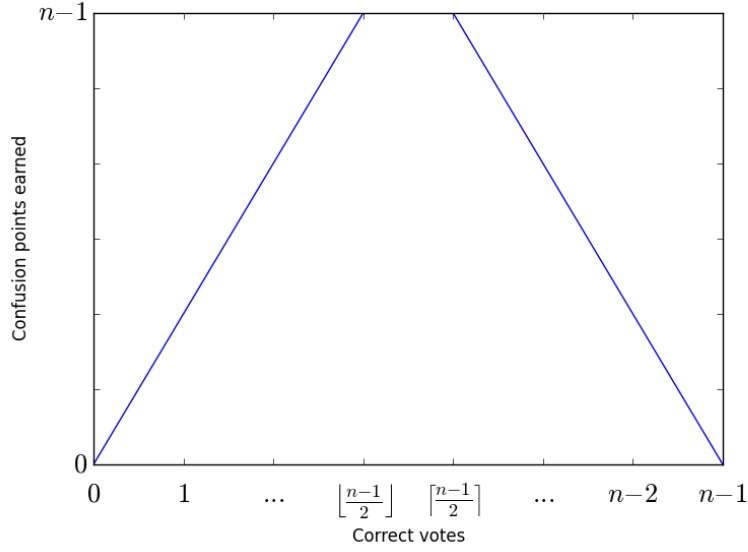


Figure 3.5: Earned confusion points in dependance of the number of correct votes. n denotes the total number of players.

3.2.3 Missing Title Penalty

As stated in Section 3.1.1, there are cases where a game round might start even if there is only one non empty video queue. As we want to avoid having passive players who participate in the game without adding videos and thereby presenting content, players with empty video queues get a penalty. When picking a player to select a video from at the start of a game round, the game server checks if there are participating players with an empty video queue. Those players will get a 4 points missing title penalty after the *Voting Phase*.

3.3 Game Speed

A game round can be very fast (in case all players choose to skip and make fast votes), it might only take about 30 seconds. The most stressful part for players

is to watch their video queues and make sure they always contain at least one video. This is one reason for the *video timer* which was introduced in the previous subsection. By default, a game round takes $\max(60 + 20, \text{video timer})$ seconds. The first number in the function might be even lower as the *voting timer* is set to $\max(20, \text{voting timer})$ seconds after the the first two votes have been made and to 0 seconds as soon as all players made their vote. So the length of the video will be the dominating factor in most cases as most videos are longer than 30-40 seconds. This gives players some additional time to find videos to put into their video queues. As soon as they found a suitable video, they can click *skip* and thereby confirm that they are ready to play the next round.

As stated above and in Section 3.1.1, the *Voting Phase* can only take another 20 seconds after the first two votes have been made. If we required only one vote in order to set the *voting timer* to 20 seconds, the uploader of the current video might always make his vote as soon as possible as this puts a lot of time pressure on the other players. This strategy might reduce their chance in identifying the uploader. We considered that such behaviour might as well reveal the uploader to his co-players, but as we think that the possibility for one player to reduce the remaining voting time drastically allows more abuse than benefit, we still decided to reduce the voting time only after two votes have been made.

3.4 Information Sharing

During the *Voting Phase* of the game, the general principle is that players only see their own vote. There is however a relaxation of this rule. As soon as the second vote is made, players see who has already made his vote. This ought to spice up things a little bit. For example, players see if they always have to wait for one specific player, or a player knows if he has to hurry up in order to have a chance to earn a decent amount of points. The votes are displayed after the second vote in order to allow the uploader to make his vote early and then to enjoy the rest of the video without revealing himself to the other players.

3.5 Bots

So far we explained how human players interact with the game. But as an additional challenge for those players we added bots to the game. In every game round there is a bot among the players. It behaves just like a human player, i.e. it adds videos to its video queue and it votes for a player in each game round. Which player it votes for is a random choice, as is the point in time when it makes its vote. However, it is never too late, i.e. it never gets a negative amount of voting points.

There are two different bots: the random bot and the copycat bot. They differ

in how they choose a video to add to the video queue. Which bot plays in a game round is decided uniformly at random, i.e. each bot plays with a probability of 50 percent. The two different bots will be explained in the following sections.

3.5.1 Random Bot

The random bot adds random Youtube videos to its game queue. Youtube's data API does not provide a method find a random video directly. However, there is a trick how this can be achieved. Each Youtube video has a public id. The Youtube data API offers a method that returns a list of matching full video ids when provided with a substring of an id [16]. In order to get a valid random video id, we can guess the first few characters of the id and then send a request to the API asking for matching ids. If there are matching ids, we pick one at random, else we repeat the previous step.

3.5.2 Copycat Bot

The operating mode of the copycat bot is more complex. It tries to mimic one of its co-players. This co-player is chosen at random amongst all human players participating in a game round. The bot then looks at the videos in that player's video queue and picks a random one of those to mimic. The Youtube data API offers a function that returns similar videos when provided with a video id [16]. So in order to find a similar video to the picked one, the bot can directly ask the Youtube data API. As there are duplicate videos on Youtube it might happen that the copycat bot picks a copy of a video which it tries to mimic. For now, we omit trying to catch such cases.

3.6 Observe Mode

The game is designed to be played in a group of friends. We suggest that players are at the same location when playing, as observing the reactions of colleagues when watching a video might give valuable information about the uploader's identity. But a group of friends at the same location, each of them watching a video on his personal device does not seem to be the best way how to conduct a social game. Having a central device (like a laptop connected to a beamer or a large screen) that shows videos and scores to all players would allow more interaction between the players and allows them to watch videos together instead of each separately on his device. Enabling such a scenario is the main purpose of the observe mode.

As soon as a game is running, it can be joined in observe mode. A user in observe mode sees exactly the same as a player, but he can not make votes and

does not earn points. A central device can join the game in observe mode in order to serve as a video and scores display. The observe mode can also be used by interested users to watch the game without actively participating.

3.7 Rating Videos

As one purpose of the game is to let players get to know new content it makes sense to remember the good part of that content. That is why players can rate the picked videos. If more than half of the players like a video by uprating it, then that video will be added to the private playlist in which the game takes place. That way, the private playlist acts as a memory of the game. However, the rating of a video does not influence the points earned by the uploader.

Application Design

We created an application to test and evaluate both our design of cooperative playlists and the game. As being available to as many people as possible is of prime importance in order to keep the potential user base large, we decided to create a web application. The primary advantage of that approach is that almost each platform offers a browser. That way the application could possibly be available for all smartphones, tablets, notebooks, and PCs without us having to create platform specific implementations.

However there is a disadvantage in choosing a web application when compared to native applications for apple smartphones. Videos can not be played automatically in iOS browsers [17], which means that iOS users can only listen to cooperative playlists if they click a play button every single time a new video is loaded.

4.1 Libraries Used

4.1.1 Server Side

Each device running our application communicates with our web server. The web server runs on "Flask", a micro framework for python [18] which is based on "Werkzeug" and uses the "Jinja 2" template engine. Data is stored in a MySQL database using SQLAlchemy [19]. SQLAlchemy is a SQL toolkit and an object relational mapper for python. It can be used for many SQL based databases like MySQL or SQLite.

The playlists and the game rely on updates from the server upon certain events, like for example a vote from a player in a game or a user rating for a video in a playlist. We use Socket.IO [20] to accomplish that. Socket.IO is a javascript library for real time applications. It supports several transporters (like WebSocket, Adobe Flash Socket, AJAX Long Polling) and selects the most capable one at runtime [21]. By using Socket.IO we do not rely on constant polling by the client. FlaskSocketIO [22] is an adapter that allows to use the Socket.IO

API within "Flask" applications, which we use in our server side implementation.

4.1.2 Client Side

Socket.IO has two parts: a server side library (which was explained in the previous section) and a client side library. FlaskSocketIO has only proven stable when used with client side Socket.IO up to version 0.9.16 [22] (current: 1.3.5, checked on 29.08.2015). This is a disadvantage, however everything we needed was available in version 0.9.16.

In order to get a relatively cheap mobile version of our application we use "Bootstrap" [23] as front end framework. "Bootstrap" is configurable via less files and pre styles most html elements. It also adds additional html constructs like button groups and offers a lot of javascript functionality, from which we only used a small subset for our application. "Bootstrap" relies on jQuery [24] which we use anyway.

The website contains several lists of items that are dynamically rearranged. We use the jQuery plugin MixItUp [25] to show smooth animations when rearranging a list. There are occasions where we want to display charts. Chartist.js is a fully responsive highly customisable chart library for javascript [26] which we use for that purpose. "Mustache" [27] serves as a javascript template engine.

4.2 Connectivity

Users have to be connected to the internet if they want to watch a playlist or to play the game. As the content that is played has to be loaded from the internet anyway this does not really impose an additional requirement. However, the connection that is handled by Socket.IO must persist as long as a user wants to watch a playlist or to play the game, i.e. as soon as this connection gets interrupted the user is thrown out of the playlist and has to reconnect.

4.3 Authentication

Each user has to create a login. When creating a login, a user has to specify a username, an email and a password. The username and the email address have to be unique over the whole set of users. Currently there is no purpose for storing the email address, but in prospect of functions like "forgot password" or "forgot username" we decided to store it anyway as it enables to introduce such functions later. A user has no access to any part of the application without being logged in.

4.4 Views

In this part we describe the appearance of the web application and how users can interact with it. The application has three main views: the Lobby, the Playlist and the Game. Each of those views will be set out in detail in the following sections.

4.4.1 Lobby

The Lobby is where users can manage their playlists. They can create and join playlists and they can filter and sort the existing playlists by certain attributes. Figure 4.1 shows a screenshot of the Lobby. The following paragraphs will all refer to that figure if not stated otherwise.

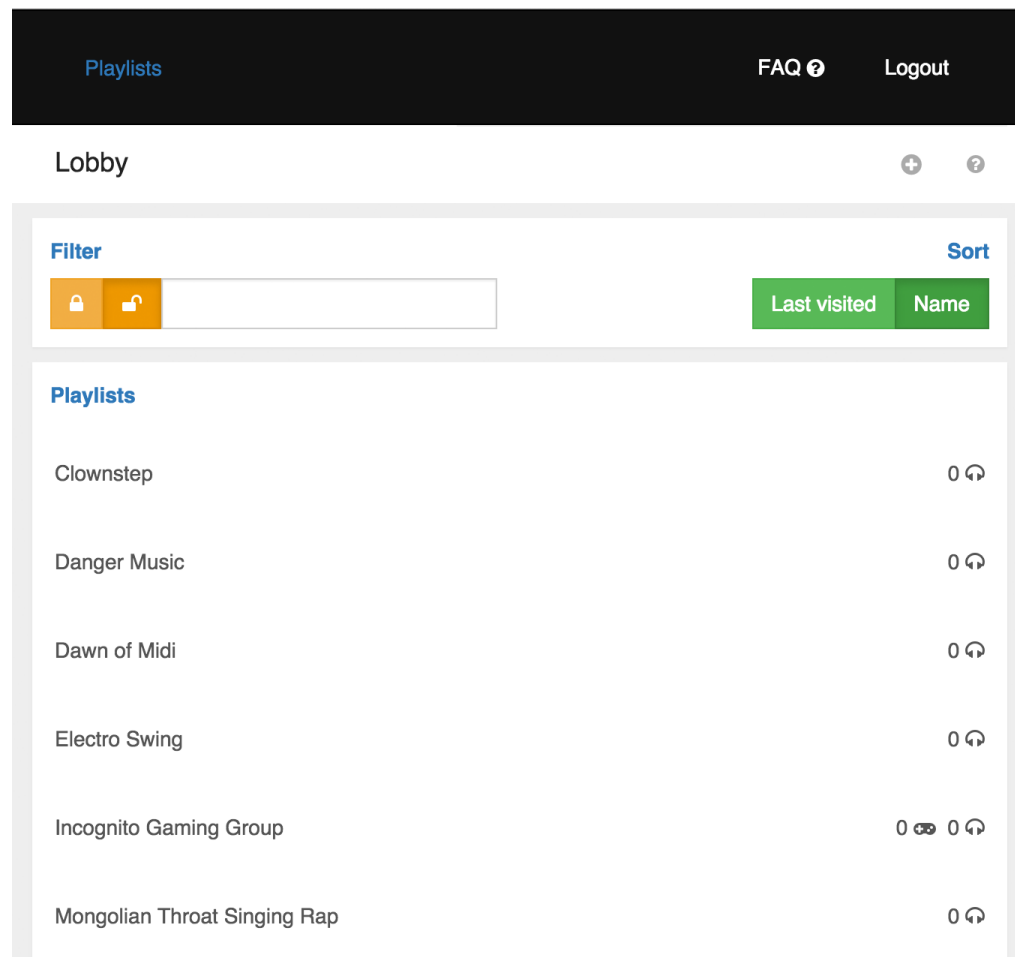


Figure 4.1: A screenshot of the Lobby of our website.

Main Menu

At the top, there is the main menu with three entries, namely Playlists, FAQ and Logout. Playlists is currently active, it is a link to the Lobby. FAQ links to a page containing a short manual for the site. Logout logs out the currently logged in user.

Sub Menu

Below the main menu there is the sub menu. It shows view specific options and in the case of Lobby it shows two icons, a plus and a question mark. The question mark is present in every view and opens a pop up containing a view specific manual. The plus icon opens a pop up that allows to create a new playlist. This pop up is shown in Figure 4.4.

Filter and Sort

The filter and sort widget allows to filter and sort the playlists shown in the list below. There are two filter criteria: the name of the playlist and if it is locked or not. A playlist is locked if it is private and the user has not joined it, i.e. he did not provide the secret passphrase. All other playlists are unlocked. There are two attributes by which the playlists can be sorted: name and last joined. Last joined is the time when the playlist was last joined by the user.

Playlists

Below the filtering and sorting widget there is a list of the filtered playlists. The playlists in Figure 4.1 are unlocked. There are both private and public playlists. Public playlists show the number of current watchers on the right side. Private playlists show both the number of watchers and the number of *In Game* players on the right. A playlist can be joined by clicking on its list entry.

Locked playlists have a different behaviour. Figure 4.2 shows a list containing locked playlists. When clicking on an playlist entry a pop up appears, asking for the secret passphrase in order to join the playlist. This pop up is shown in Figure 4.3. The passphrase will be remembered and will not have to be supplied again.

Creating a new Playlist

Figure 4.4 shows the pop up that appears when creating a new playlist. The user has to specify a name and in case he wants to create a private playlist, a secret passphrase.

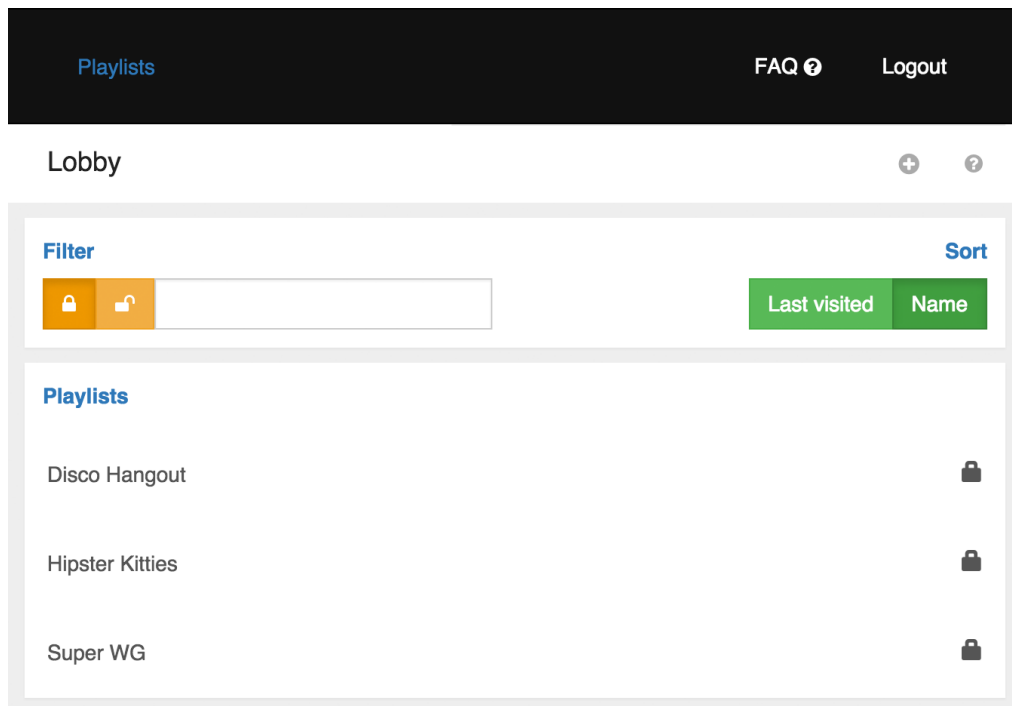


Figure 4.2: A screenshot of the lobby displaying locked playlist entries.

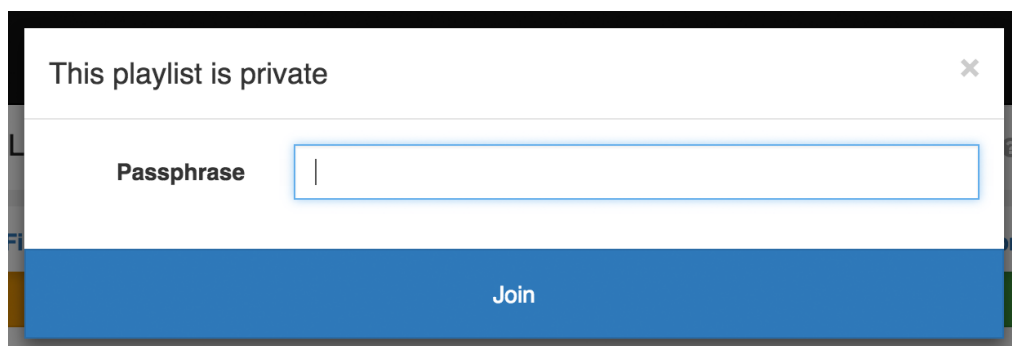


Figure 4.3: A screenshot of the pop up that appears when trying to join a locked playlist.

4.4.2 Playlist

As soon as the user joins a playlist he sees the playlist view. Figure 4.5 shows a screenshot of this view. The following paragraphs will all refer to that figure if not stated otherwise.

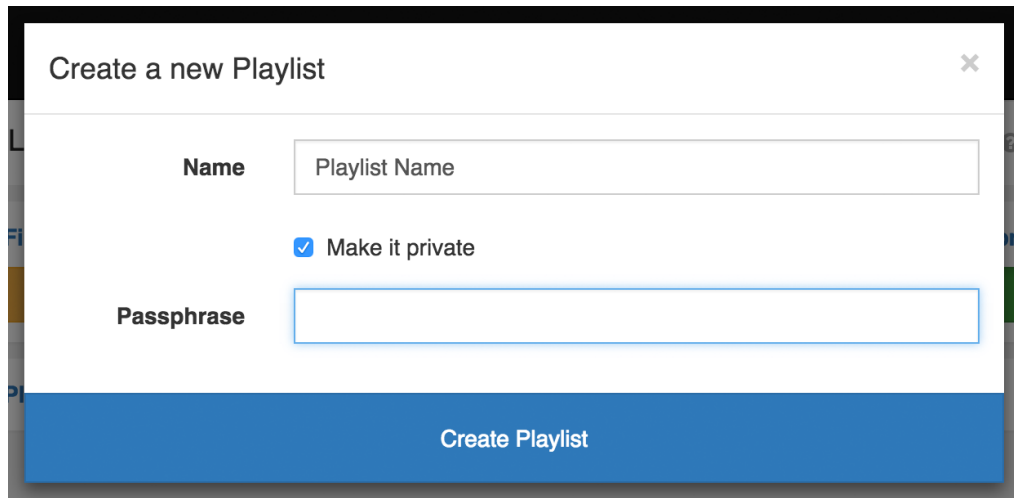


Figure 4.4: A screenshot of the pop up that allows to create a new playlist.

Sub Menu

At the top there is the sub menu. On the left, it shows the title of the playlist and also in which mode the user is listening (either *RADIO* or *DICTATOR*). Besides the question mark, there are two other icons: a trophy and a switch. The trophy opens a pop up showing the scores in the game of that playlist. This icon is only present in private playlists. The switch toggles between the two listening modes *Radio* and *Dictator* upon clicking.

Youtube Player

Below the sub menu, there is the iFrame Youtube player. It offers built in features like setting the volume or play/pause. Videos are automatically played using the Youtube iFrame Player API. As mentioned earlier in Section 4, this does not work for iOS devices. At the bottom of the player, there is an input field to add new videos to the playlist. The user can paste the URL of a Youtube video to this input in order to add it.

Current Video

The current video widget is right below the Youtube player. It shows the title of the current video and the user can rate the video and skip. The exact behaviour was described in Section 2.3.

Playlist Titles

At the bottom there is the list of all videos in the playlist. Each video entry shows the name of the video on the left and the rating by the user on the right. The user can click on the icons in order to rate a video. The list is dynamically

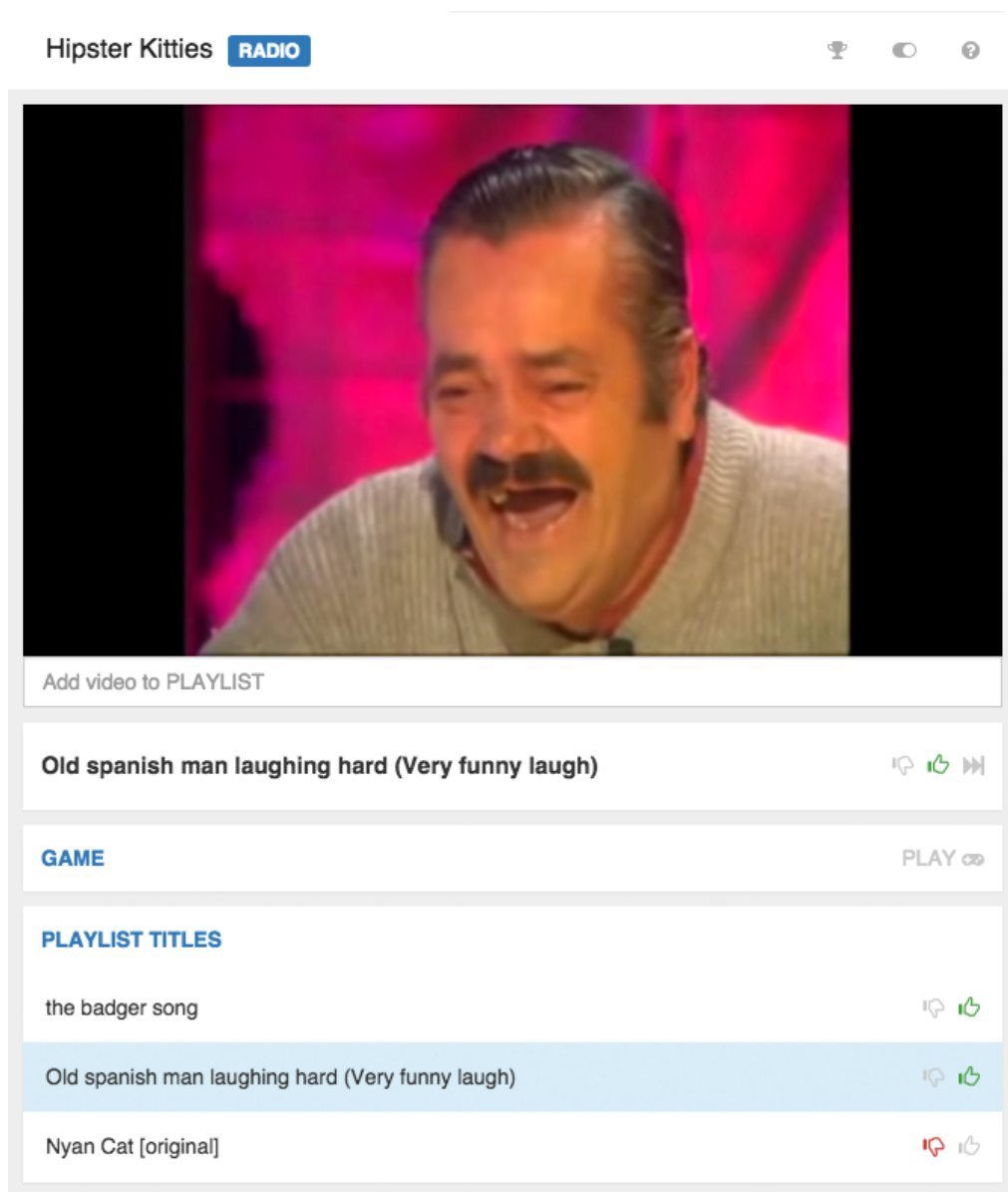


Figure 4.5: A screenshot of the playlist view. The displayed video is taken from Youtube [28].

rearranged as soon as the order of titles changes due to user ratings or playing the next video.

Game Tab

The game tab is either open or closed. In Figure 4.5, the game tab is closed. It can be opened by clicking *play*. Figure 4.6 shows the open game tab. The

input allows to add videos to the personal video queue. The user can paste the URL of a Youtube video there. The number of videos currently present in the video queue is displayed on the right. Below there is a list of all players who are not in state *Idle*. We use a color scheme to distinguish the state of each individual player.

blue: The player is currently in state *In Game*.

green: The player is *Ready*, i.e. he has both clicked ready and at least one video in his video queue.

orange: The player is between the states *Interested* and *Ready*, i.e. he has either clicked ready or at least one video in his video queue but not both.

red: The player is in state *Interested*.

The game tab contains two to three buttons: the *play* button, the *ready* button and in some cases the *observe* button. The *play* button toggles between the two game tab states open and closed. When the game tab is closed, the user is in state *Idle*. The *observe* button is only present if the game is not in state *Idle*. As the game in Figure 4.5 and Figure 4.6 is in state *Idle*, the *observe* button is not shown there. Figure 4.7 shows the game tab when the game is running.

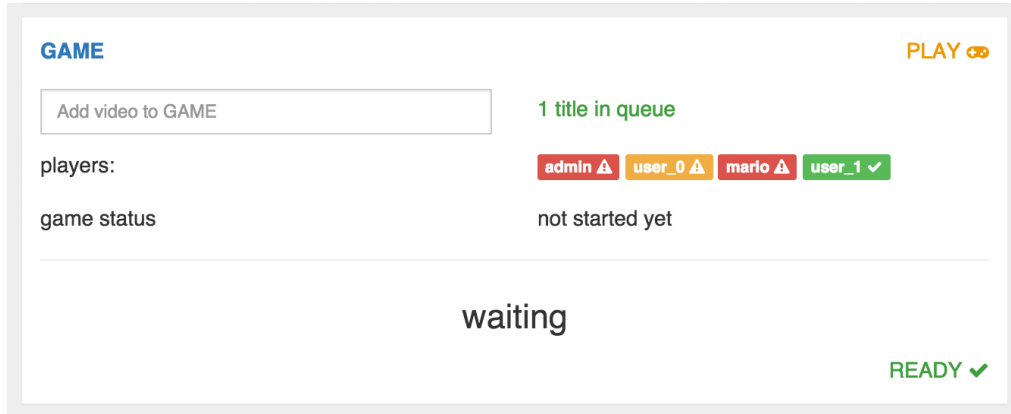


Figure 4.6: A screenshot of the open game tab.

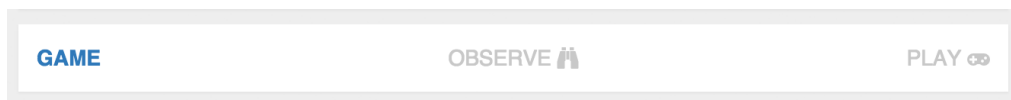


Figure 4.7: A screenshot of the game tab showing the *observe* button.

4.4.3 Game General

The game view is very similar to the playlist view. There are many elements that are present in both views, sometimes in a slightly different version. Figure 4.8 shows a screenshot of the upper part of the game view. This part is the same in *Voting Phase* and *Scores Phase*.



Figure 4.8: A screenshot the upper part of the game view. The displayed video is taken from Youtube [29].

Sub Menu

On the left there is the title of the playlist as well as an indicator in which mode the user is watching or attending the game (either *GAME* or *OBSERVE*). Besides the question mark, there are two additional icons. The trophy icon has the same function as in the playlist view. The Youtube player icon toggles the Youtube player between shown and hidden. This function is meant for use in combination with the observe mode (see Section 3.6) as there is no need to show the Youtube player if there is a central display.

Youtube Player

The Youtube player shows the picked video of the current game round. Its functions are exactly the same as in the playlist view.

Current Video

The current video widget shows the title of the picked video as well as up and down rate buttons and a skip button. The functionality of the skip button was explained in Section 3.1.1, the one of the rating buttons in Section 3.7.

4.4.4 Game State Specific - *Voting Phase*

Figure 4.9 shows a screenshot of the bottom part of the game view when the game is in *Voting Phase* and the user in *In Game Vote Not Made*.

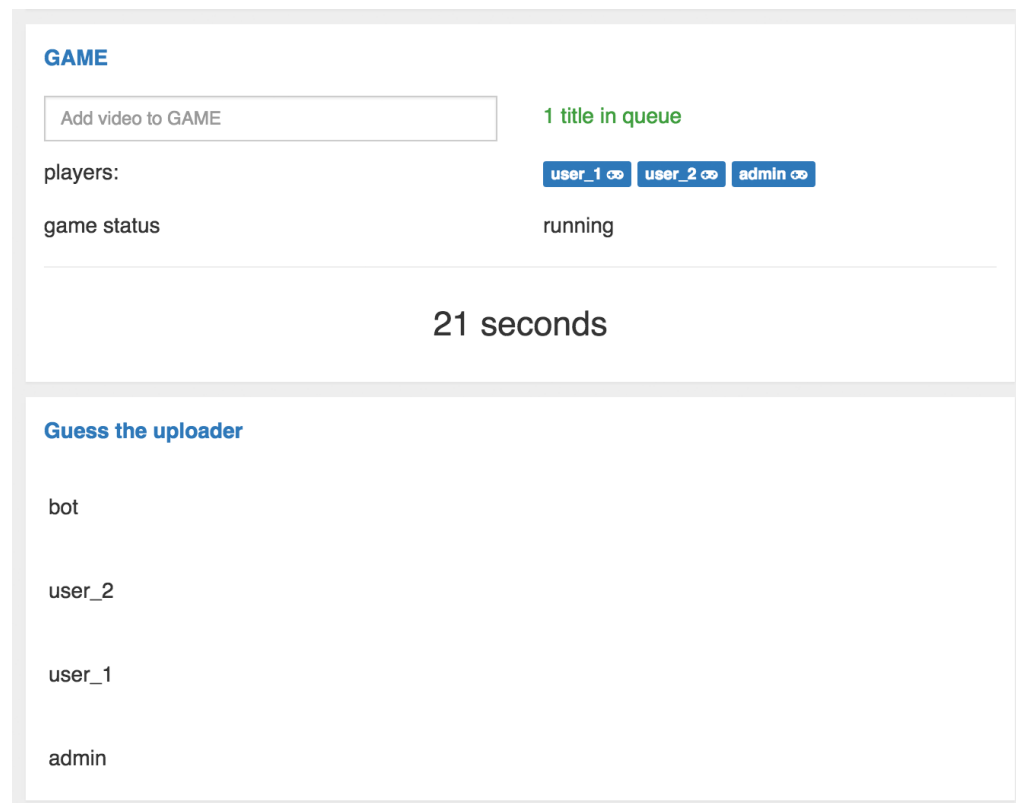


Figure 4.9: A screenshot the bottom part of the game view in *Voting Phase* with the user in *In Game Vote Not Made*.

Game Tab

The game tab shows the same information as in the playlist view. The only

difference is that there are no buttons and that there is a timer showing the remaining duration of the game round's *Voting Phase*.

Uploader List

The uploader list contains an entry for each possible uploader of the shown video. The user has to select an entry in order to make his vote. Figure 4.10 shows a screenshot of the bottom part of game view after the user made his vote. The user's vote is marked in blue and on the right of each entry there is an orange label indicating whether that player already made his vote. As explained in Section 3.1.1, this information is displayed to all players after the first two players made their votes. As the remaining time is shortened to 20 seconds at that point, the background flashes red to signal users that they have to hurry up with making their votes. The background flashes red every two second as soon as there are less than 7 seconds remaining.

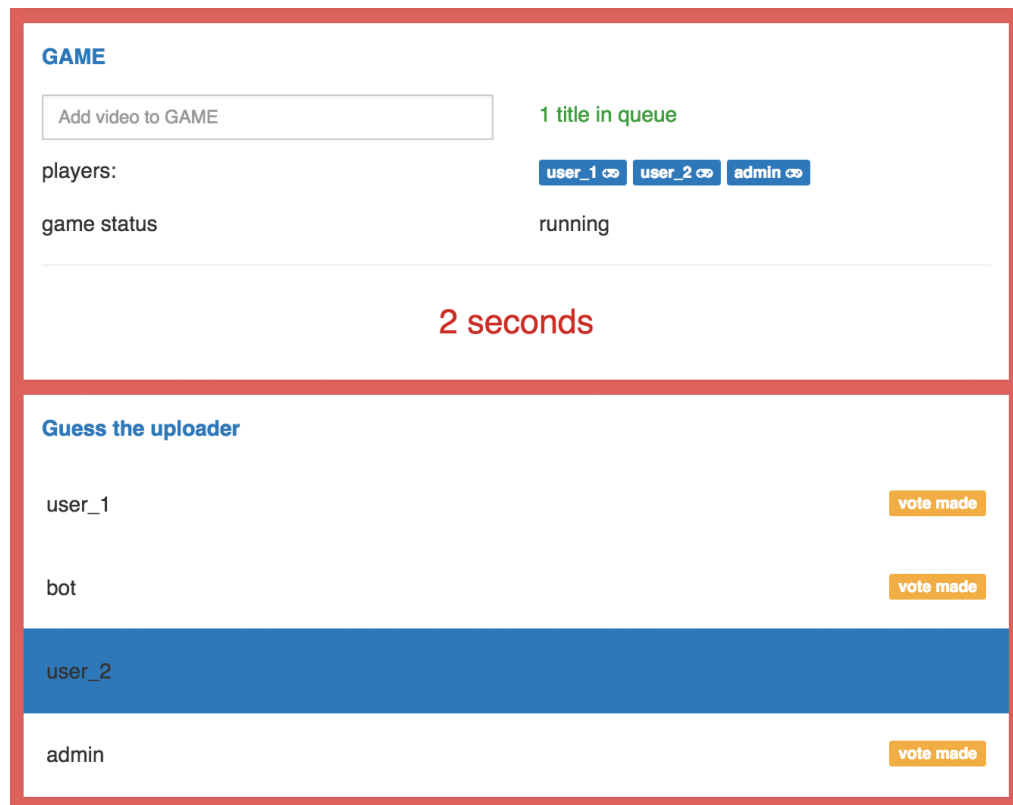


Figure 4.10: A screenshot the bottom part of the game view in *Voting Phase* with the user in *In Game Vote Made*.

4.4.5 Game State Specific - *Scores Phase*

Figure 4.11 and Figure 4.12 together show the bottom part of the game view when the game is in *Scores Phase* and the user is in *In Game Scores Phase*. It is split into two screenshots due to its large size.

GAME EXIT

Add video to GAME

1 title in queue

players: user_1 user_2 admin

game status next round starting soon

537 seconds

GAME ROUND SCORES

SOLUTION: user_2

Rank	Player	Speed	Vote	Points
1.	user_1	3.	user_2	3 + 0 + 0 = 3
1.	user_2			0 + 3 + 0 = 3
2.	admin	1.	bot	0 + 0 + 0 = 0
2.	bot	2.	user_1	0 + 0 + 0 = 0

Figure 4.11: A screenshot the bottom part of the game view in *Scores Phase* showing the game tab and the game round scores.

Game Tab

The game tab looks the same as in *Voting Phase* except for the *exit* button on the top right. The player can click this button in order to leave the game and to return to watching the playlist. Also the counter does not show the remaining duration of the *Voting Phase* but rather of the *Scores Phase*.

Game Round Scores

The game round scores widget shows the uploader of the played video as well



Figure 4.12: A screenshot the bottom part of the game view in *Scores Phase* showing the game chart.

as the votes, ratings and achieved points of all players. The points are split into three different kinds of points: voting points, confusion points and missing title penalty. Those are explained in detail in Section 3.2.

Game Chart

The game chart offers a visual illustration of each player's points over the last few game rounds. On the top right of the chart, there are different time frames to choose from, currently that is 15 minutes, 30 minutes, 1 hour and 6 hours. In each time frame, the points of each player start from zero, i.e. points made before the lower bound of the selected time frame are not taken into account.

Prototype Iteration

We did not evaluate our design of cooperative playlists as this would require a large user base and our application was not yet ready to be deployed. However, if there was a large user base we could gather data about how often users want to skip a title or how frequently the currently played video is downrated or uprated in order to infer the user satisfaction in a playlist. Based on that inferred satisfaction we could then reason whether the current design works or not.

The game was evaluated in four test sessions within the Distributed Computing Group at ETH Zürich. The six most important outcomes and resulting changes will be explained in the following part in chronological order.

Length of Game Rounds

At first we tested a version of the game in which a game round consisted of showing a video of each player, collecting the votes of each player for each of those videos and finally showing the scores. That way a game round was about n times as long as it is in the current version (where n is the number of players). As the scores were only shown after playing a video from each player, the game flow was much slower and we decided to create a game version where a round consists of picking one video and showing the scores immediately after the *Voting Phase*.

Observe Mode

After playing the game in a group of people, it became obvious that having a central display to play the picked videos would increase the fun massively as it would allow the group to watch the videos together instead of each player on his personal device.

Missing Title Penalty

Before introducing the missing title penalty, the *Voting Phase* could not start unless each player had at least one video in his video queue. This turned out to slow the game down a lot and we decided to start the *Voting Phase* if at least

one player has at least one video in his video queue. To penalise players who fail to keep their video queue non empty, we introduced the missing title penalty.

Delaying *Vote Made* Information

As explained in Sections 3.3 and 3.4, two votes are required in order to shorten the remaining duration of the *Voting Phase* and to display the information about who already made his vote. At first, only one vote was required to trigger those two actions. But as this would allow the uploader to decrease the remaining time of the *Voting Phase* significantly, we chose the set the required number of votes to two. Also requiring two votes before displaying the information about which player already made his vote allows the uploader to make his vote early and then to enjoy the rest of the video without risking to reveal himself.

Picking a Title

In order to select a player to pick a title from, we first used a random number generator without memory of previous picks. Therefore, in the evaluation there were players that were never selected to pick a title from within 20 game rounds. The crew consisted of 7 players and the probability of that scenario is shown in Equation 5.1. To avoid such behaviour, we implemented the biased random algorithm as explained in Section 3.1.2.

$$\left(1 - \frac{1}{7}\right)^{20} = \left(\frac{6}{7}\right)^{20} = 0.0458 \Rightarrow 4.58\% \quad (5.1)$$

Skipping

The current version of the game has a skip function as explained in Section 3.3. Before we introduced this function, the *Scores Phase* always lasted 20 seconds which meant that a player often had only 30-40 seconds to find a new video to add to his video queue. This turned out to be very stressful for the player whose video was picked and played. In addition there were videos we wanted to watch in full length as they hit our taste. For that reason the *Scores Phase* lasts at least as long as the video takes to finish playing unless **all** players choose to *skip*.

Bibliography

- [1] : Reddit about. (Accessed: 03.09.2015) <https://www.reddit.com/about/>.
- [2] : Imgur about. (Accessed: 03.09.2015) <http://imgur.com/about>.
- [3] : Spotify collaborative playlists. (Accessed: 02.09.2015) <https://news.spotify.com/au/2008/01/30/collaborative-playlists/>.
- [4] : Pandora radio. (Accessed: 03.09.2015) <http://www.pandora.com/about>.
- [5] : Music Genome Project. (Accessed: 03.09.2015) <https://www.pandora.com/about/mgp>.
- [6] : Plug DJ how to. (Accessed: 02.09.2015) <http://support.plugin.dj/hc/en-us/articles/203263577-I-want-to-DJ-How-do-I-do-it->.
- [7] : Plug DJ meh algorithm. (Accessed: 02.09.2015) <https://blog.plugin.dj/2012/08/the-new-meh-algorithm/>.
- [8] : Rdio subscription. (Accessed: 24.08.2015) <http://www.rdio.com/home/de-ch/#pricing>.
- [9] : Amount of content on Rdio. (Accessed: 24.08.2015) <http://www.rdio.com/about/>.
- [10] : Spotify subscription. (Accessed: 24.08.2015) <https://www.spotify.com/ch-de/premium/>.
- [11] : Spotify press information. (Accessed: 24.08.2015) <https://press.spotify.com/au/information/>.
- [12] : Youtube IFrame API. (Accessed: 24.08.2015) https://developers.google.com/youtube/iframe_api_reference.
- [13] : Youtube Data API. (Accessed: 24.08.2015) <https://developers.google.com/youtube/v3/getting-started>.
- [14] : How Not To Sort By Average Rating. (Accessed: 24.08.2015) <http://www.evanmiller.org/how-not-to-sort-by-average-rating.html>.
- [15] : The badger song. (Accessed: 02.09.2015) <https://www.youtube.com/watch?v=auED92iFVJE>.

- [16] : Youtube Data API Search Functions. (Accessed: 27.08.2015) <https://developers.google.com/youtube/v3/docs/search/list>.
- [17] : Apple iOS HTML5 Audio and Video. (Accessed: 29.08.2015) https://developer.apple.com/library/safari/documentation/AudioVideo/Conceptual/Using_HTML5_Audio_Video/Device-SpecificConsiderations/Device-SpecificConsiderations.html.
- [18] : Flask microframework. (Accessed: 29.08.2015) <http://flask.pocoo.org/>.
- [19] : SQLAlchemy. (Accessed: 29.08.2015) <http://www.sqlalchemy.org/>.
- [20] : Socket.IO. (Accessed: 29.08.2015) <http://socket.io/>.
- [21] : Gevent SocketIO. (Accessed: 29.08.2015) <http://learn-gevent-socketio.readthedocs.org/en/latest/socketio.html>.
- [22] : Flask Socket.IO. (Accessed: 29.08.2015) <https://flask-socketio.readthedocs.org/en/latest/>.
- [23] : Bootstrap front end framework. (Accessed: 29.08.2015) <http://getbootstrap.com/getting-started/>.
- [24] : jQuery. (Accessed: 29.08.2015) <https://jquery.com/>.
- [25] : MixItUp. (Accessed: 30.08.2015) <https://mixitup.kunkalabs.com/>.
- [26] : Chartist.js. (Accessed: 29.08.2015) <https://gionkunz.github.io/chartist-js/>.
- [27] : Mustache template engine. (Accessed: 29.08.2015) <https://mustache.github.io/>.
- [28] : Spanish comedian laughing. (Accessed: 02.09.2015) <https://www.youtube.com/watch?v=fyETHSM4JnI>.
- [29] : Anchorman best scenes. (Accessed: 02.09.2015) <https://www.youtube.com/watch?v=QQUi6Oi8SLQ>.