Bachelor Informatica

Universiteit van Amsterdam

Music composition in a distributed collaborative environment

Ino de Bruijn

August 5, 2009

Supervisor(s): Robert Belleman (UvA)

Signed:

Abstract

The Internet connects people. It provides the technology to collaborate on tasks even though its participants are geographically distributed. Well accepted portals now exist on the Internet for almost every imaginable form of human expression, except one: music composition. This thesis deals with the task of performing music composition in a distributed collaborative environment. A study on this subject has been done to provide an insight on what the current status of this field is. This thesis adds a new solution. The design and implementation of a collaborative environment for music composition is discussed and the results of various experiments on the system are presented.

Contents

CHAPTER 1

Introduction

The Internet brings together people from all over the world. It allows us to collaboratively work on a single task from geographically distributed locations. Various portals for human expression have emerged over the years on the Internet. For instance: there are portals for information, such as webpages and wikis, and social portals like Hyves[1] and Facebook[2]. Other examples are portals for personal voice such as Skype[3] and portals for public voice such as a podcasts. There are also portals for text: personal text can be written in email or by using instant messaging and public text can be written on Twitter[4]. The list goes on, but nowhere in the list can a portal for music composition be found. The task of creating such a portal for music composition is the task this research revolves around.

There are several ways of creating a music composition with multiple artists. A well-known formula is creating songs by jamming. This can be done by multiple artists playing an instrument at the same geographical location. However, if the artists are at geographically distributed locations it would be impossible to hear each other, because of the fundamentals of sound. A possible solution to the problem of having artists at geographical distributed locations is by letting them jam over the Internet. This is bound to certain time constraints. In order to interpret two sounds as simultaneous, one must hear both sounds within a time frame of 20 milliseconds.[1, 2] Nodes in a network cannot send information to other nodes in a network faster than the speed of light which limits the distance that can be traveled in 20 milliseconds. If we wanted to perform music composition without this limit in distance, we would have to think of a different solution. The research question in this thesis is to provide an alternative solution to the problem of performing music composition in a distributed collaborative environment.

The first chapter gives a complete description of the problem and gives an overview of related work on this subject to place the research into a certain context. In the following chapters we will look at the design and the implementation of our solution to the problem. Then the results of some experiments on the system will be presented. The final chapter presents our conclusions.

[1] Hyves `http://www.hyves.nl/`
[2] Facebook `http://www.facebook.com/`
[3] Skype `http://www.skype.com/`
[4] Twitter `http://twitter.com/`

CHAPTER 2

Problem description and related work

2.1 Problem description

The problem addressed by this thesis is to design, implement and test a collaborative environment that allows musical artists from geographically distributed locations to work together on a music composition. This collaborative environment is a shared space that supports access by multiple users simultaneously. When researching different possibilities for such an environment the following questions should be taken into consideration. How many users are supported? Does the solution scale with an increasing number of users or is it limited to x users? What should be done when multiple users "compete" for access to the same composition with regard to the problem of being able to work together?

We constrain several aspects of the problem. Music compositions are stored as sound files ("samples") instead of written down as musical notation. The composing is done through "layering" music tracks on top of each other on a time line. This is a familiar model to a lot of musical artists. Finally, network access over the Internet is used to provide support for geographically distributed locations.

2.2 Related work

This section provides an overview of current solutions to the problem of performing music composition in a distributed collaborative environment. They roughly fall into two categories. There are the ones that offer collaborative music composition through Internet jamming, which differ mostly in the way that they deal with latency issues. The other category offers collaborative music composition by uploading sound files. The allowed audio formats, storage space and the user interface are the distinctive factors here.

2.2.1 NetJam

NetJam is probably the earliest attempt to collaborate on music compositions on the Internet. Craig Latta states in his article *Notes from the NetJam Project* that 'NetJam is a computer network that provides a means for people all over the world to collaborate on music compositions and other works of art.'[3] It is basically a mailing list people can subscribe to. Someone can change the song, which is in a Musical Instrument Digital Interface file (MIDI)[1] format, and e-mail it to all the other Netjam participants. By following certain rules, like only editing your own MIDI channel, the composition stays well-ordered. NetJam is not maintained currently, but the author can still be contacted about it through his website.[2]

[1] Manufacturers Association *Standard MIDI Files Specifcation* `http://www.midi.org/techspecs/smf.php`
[2] Craig Latta *NetJam* `http://netjam.org`

2.2.2 NINJAM

The Novel Intervallic Network Jamming Architecture for Music is an application that offers a form of Internet jamming.[3] It works with a central server that sends the sound of each client in Ogg Vorbis format to all other connected clients.[4] Their solution to the latency problem is to measure this latency in measures. Each client plays x measures where x is bigger than the largest found latency. The sound that was recorded in these measures are then send to all other clients through the central server. This way you are always hearing the sound that the other clients made in the previous measures.

2.2.3 eJamming AUDiiO

The commercial eJamming AUDiio is another application for Internet jamming.[5] It says it is able to let musicians whom are 1000 miles apart play together with zero latency. The algorithms they use are not available for the public eye, but they appear to be delaying your own sound in order to sync it with the other clients. The sound is sent between clients using peer-to-peer technology. This system works very well if the latency is low. When the latency becomes higher the interaction with your own instrument becomes troublesome. You can then however still make songs together. eJamming offers a way to make compositions by letting everyone record a track of a song in turns.

2.2.4 GridJam

Gridjam is an experimental multimedia event that brings together visual artists, composers, musicians and computer scientists from geographically distributed locations.[6] The event utilizes the LambdaRail[7] network, a high speed network used for sharing large data sets between researchers around the globe. The network is used to demonstrate a live, partly improvised, 3D visualized musical performance. It is totally different from the previous on line jamming tools in that it does not try to be a on line jamming tool, but rather an instrument. It sees latency not as an issue, but as part of the music much like rests between notes and phrases in Western music.[4] Musicians interact with each other through the visuals they trigger on the 21^{st} Century Virtual Color Organ. This organ translates music compositions into visual performance. These music compositions consist of MIDI data. Currently GridJam is still in development.

2.2.5 Sites on the World Wide Web offering collaborative music composition

There are various sites on the World Wide Web that offer a way to share samples and ideas. Among the best-known are cocompose.com[8], kompoz.com[9] and myonlineband.com[10]. They all have pretty big communities with several songs that are worth listening to. Songs can be created with a specified group of people or can be created with anyone. Some allow entire Cubase or Ableton compositions to be stored while others only allow audio formats such as mp3 and wave. All offer some free storage space. Cocompose also offers the feature to create compositions out of tracks in the same way that eJamming AUDiiO does. Users can create songs together by recording on top of an already existing tune.

2.3 What does this research contribute?

The solutions that fall in the category of music composition through online sound files storage are all still very limited. One problem is that none of the user interfaces provide a way to

[3] NINJAM `http://www.ninjam.com/`
[4] Xiph.org *Ogg Vorbis I specification* `http://xiph.org/vorbis/doc/Vorbis_I_spec.html`
[5] eJamming AUDiiO `http://www.ejamming.com/`
[6] GridJam `http://jackbox.net/pages/gridjampages/Gridjam1.html`
[7] LambdaRail `http://www.nlr.net/`
[8] Cocompose `http://www.cocompose.com`
[9] Kompoz `http://www.kompoz.com`
[10] My Online Band `http://myonlineband.com`

arrange stored samples to be played sequentially in a composition. Compositions can only exist of audio files that are played at the same time. This requires the user to upload an entire track, a sample which length is equal to the length of the entire composition, each time they want to add something new to the composition. This is costly for the server, because larger samples than necessary have to be stored. A different problem is that composing is always limited to one person at a time. Editing a composition with multiple people at the same time could increase the interaction between the musical artists working on that piece. The time to create a composition could be reduced as well. This research tries to provide a solution for both problems.

CHAPTER 3

Design

In the previous chapter the problem of this thesis was formulated as providing a collaborative environment that allows musical artists from geographically distributed locations to simultaneously work together on a music composition. The design of the system that provides this environment will be outlined in this chapter. First the requirements for the system are explained which were the decisive factors in choosing among various design options for each component of the system. Then the design of all the individual components is explained. We end with an explanation on how all the components work together.

3.1 Requirements for the system

In order to be able to make a choice between the various design options the following requirements were set up:

1. A music composition consists of multiple samples either placed parallel to each other on different tracks or in sequence with each other on the same track. Not required, but preferred: music compositions consisting of samples placed at any given point on a time line.

2. A sample is an audio file in either the Motion Picture Expert Layer Group-1 Layer 3 (MP3), Ogg Vorbis (Ogg) or Waveform audio format (WAV). These are popular formats that most users will be familiar with. This allows musical artists to create samples for the system using all popular recording software.

3. The volume of every sample can be changed individually.

4. A music composition that has been created with the software is audible. The ability to play a composition outside of the system using standard players is a plus.

5. The system provides network access over the Internet to support access from geographically distributed locations.

6. The system is scalable. If the system becomes popular it should be able to deal with the increase in user numbers.

7. The requirements for a user to use the system should be low, so that it reaches a broad audience.

8. The system can deal with multiple people editing the same music composition.

9. The music compositions are persistent. This way the music compositions can be edited from distributed locations in time.

10. The provenance of a music composition is saved. In case the music composition is changed in an undesirable way, the music composition can be restored to a previous version.

The system designed should conform to all the previously stated requirements.

3.2 System overview

The system can be divided into several components using the model-view-controller pattern (Figure 3.1). The model represents the data that can be manipulated by the application. How

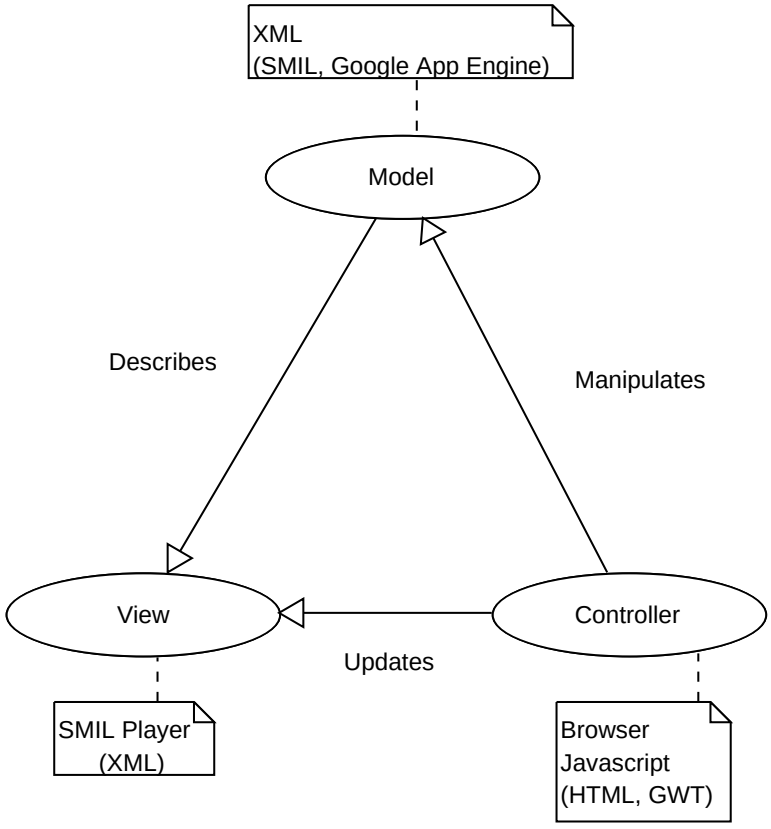


Figure 3.1: Model-View-Controller pattern

the data is stored is encapsulated in the model as well. The view is the way the model is presented to the user. The controller handles events that either update the view, manipulate the model or both.

In our design the model component consists of the music compositions (requirement 1-3) and how they are stored (requirement 6-7). The choice of the view determines how users can listen to the composition (requirement 4, 7). The controller determines how users can manipulate the composition (requirement 5-8). In the following sections the design of each component will be explained. In the final section the way all components work together is explained (requirement 8-10). The design of the model component is explained first.

3.3 Model

The model component of the system determines the representation of the data and the data access layer. We first describe the format of the data, followed by an explanation on how the data is accessed.

3.3.1 Representation of the music compositions

To represent the music compositions the Synchronized Multimedia Integration Language (SMIL) is used. SMIL is a W3C recommended XML-based language that allows authors to write interactive multimedia presentations.[5] 'Using SMIL, an author may describe the temporal behavior of a multimedia presentation, associate hyperlinks with media objects and describe the layout of the presentation on a screen.' SMIL conforms to all the requirements set up for the design of the model:

- SMIL specifies audio files to be played sequential, parallel or at a given region in time (requirement 1).
- SMIL does not constrain audio files to be of a specific audio format, therefore any format can be used (requirement 2).
- Volume levels of audio files can be specified (requirement 3).

By using SMIL the application could be extended to use video, images and text in compositions as well. For this research we will only focus on the features necessary to describe our music compositions. A SMIL presentation that plays some sounds could look like this:

```
<smil xmlns="http://www.w3.org/ns/SMIL" version="3.0" baseProfile="Language">
  <head>
    <layout>
    </layout>
  </head>
  <body>
    <seq>
      <audio src="http://sounds.com/horn.mp3" />
      <audio src="http://sounds.com/door.wav" />
      <par>
        <audio src="http://sounds.com/bell.wav" />
        <audio src="http://sounds.com/knock.wav" />
      </par>
    </seq>
  </body>
</smil>
```

This simply plays the first two sounds sequential and plays the last two sounds parallel after `door.wav` has ended. SMIL uses `<audio>` tags to specify an audio media object where the `src` attribute is an Uniform Resource Locator (URL) as can be seen in the example. The temporal behavior of the presentation is specified by using one or several:

- `<par>` tags.
- `<seq>` tags.

The `<seq>` element indicates that child elements should be played sequential and the `<par>` element indicates that child elements should be played parallel. The time to play an audio media object can also be specified in milliseconds, seconds, minutes and hours using one or several of the following attributes:

- `begin`
- `end`
- `dur`
- `repeatDur`

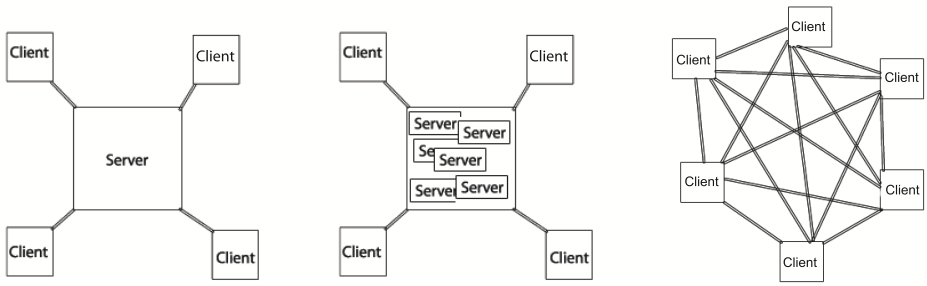
Requirement 3 that states the volume of every sample can be changed individually is fulfilled with the `soundLevel` attribute of media objects. Its value can be given in percentages or decibels. SMIL furthermore allows the grouping of samples comparable to the way audio is played on a track in multi track recording software. When the output volume of a track is changed in these programs all volume of the audio that plays on that track is altered. SMIL uses the `region` attribute to group media objects to a `region` element, which could be seen as the SMIL representation of a track. If the `soundLevel` attribute of a `region` element is set, every sample played in this region will have its volume set to the specified `soundLevel`, unless the audio media object has its own `soundLevel` attribute set. In that case the `soundLevel` will be the combination of the two. Here is another example that features times specified in seconds and playing audio media objects in regions:

```
<smil xmlns="http://www.w3.org/ns/SMIL" version="3.0" baseProfile="Language">
  <head>
    <layout>
      <region xml:id="loud" soundLevel="90%" />
      <region xml:id="soft" soundLevel="20%" />
    </layout>
  </head>
  <body>
    <par begin="0s">
      <audio begin="0s" end="5s" region="loud" src="http://sounds.com/horn.mp3" />
      <audio begin="2.5s" end="5s" region="soft" src="http://sounds.com/door.wav" />
    </par>
  </body>
</smil>
```

In the following section the way the SMIL compositions are stored is explained.

3.3.2 How are the compositions stored?

The compositions can be stored at one location or at multiple locations. To determine what data access layer would work best for our system, three different logical topologies were considered for the application based on certain frameworks that were available (Figure 3.2): the centralized client server topology, the decentralized server cloud topology and the fully connected client topology. The centralized client server topology has one server where multiple clients connect



(a) Centralized Client Server (b) Decentralized Server Cloud (c) Fully Connected Client

Figure 3.2: Considered topologies

to. The decentralized server cloud lets multiple clients connect to a cloud where one or multiple servers in the cloud handle the requests of each client. The fully connected client topology connects every client to every other client. We compare the topologies on the properties of the number of supported users, scalability, whether or not the topology has a single point of failure, the implementation effort and the maintenance effort:

Properties	Centralized Client Server	Decentralized Server Cloud	Fully Connected Client
How many users are supported?	Number of users one server can handle	Number of users one server can handle multiplied by the number of servers	Undefined, depends on implementation
Scalable?	No	Yes	Yes
Single point of failure	Yes	No	No
Implementation effort	Low	High	High
Maintenance effort	Low	High	Low
User requirements	Low	Low	High

Table 3.1: Scalability of the Topologies

The centralized client server topology can only handle a number of users equal to the number of users one server can handle. If that server fails, the entire system fails. A benefit of having one server is that only one server has to be implemented and maintained. The centralized client server topology conforms to requirement 7, because the user requirements for a user to use the system can be kept to a minimum. All processing that requires certain libraries or software can be done on the server side. However, the topology does not conform to requirement 6, because the system is not scalable. When the limit on the number of users that one server can handle is reached, either the server fails or certain client requests are ignored. A topology that does allow scaling of the system is the decentralized client server cloud topology. The client requests are distributed over several servers. When there are too many client requests for the cloud the handle extra servers can be added to the cloud. The topology does not have a single point of failure, because the tasks of a server that fails can always be taken care of by a different server in the cloud. Multiple servers have to fail before the entire system fails. A disadvantage of the topology is that implementing all the servers and maintaining them can be a troublesome process. The fully connected client topology does not have the maintainability issue, because the topology only consists of clients and does not require running multiple servers. The topology can be kept alive simply if it is used by users. Every user is connected to every other user, therefore the topology has no single point of failure. Scaling the topology only requires connecting all the current users to the new user. The problem with the fully connected client topology is that it increases the user requirements. Every user needs to run a program to connect with the other clients in the topology which is likely to use libraries not every user has. The fully connected client topology topology does not conform to requirement 7.

The only topology that conforms to both requirements 6 and 7 is the decentralized server cloud topology. The maintenance effort and the implementation effort can be reduced as well by implementing the topology using the Google App Engine[1]. Google offers this service which runs web applications programmed in either Java or Python on Google's infrastructure, an infrastructure comparable to the decentralized server cloud topology. The applications are run in a secure sandbox environment that only gives limited access to the operating system on the server. This allows them to distribute the application over multiple servers when traffic increases. Running all the servers for the second topology is therefore not an issue anymore and the time to implement the topology is greatly reduced as well. Since the decentralized server cloud topology is the only topology conforming to all the requirements, the decentralized server cloud topology is chosen as the topology for our system.

In the following section the view of the model is discussed.

[1]Google App Engine `http://code.google.com/intl/en/appengine/docs/whatisgoogleappengine.html`

3.4 View

The requirement for the view is that compositions created with the software are audible (requirement 4). The music compositions are in the SMIL format. SMIL can be played in either Quicktime[2], Realplayer[3] or Ambulant[4]. These are all players that can be embedded in a browser. Using an embedded player in a browser to play the compositions gives users only two requirements to use the system. The user should have a browser and a SMIL player installed. This conforms to requirement 7. In the best case scenario the SMIL specification is adopted by all popular browsers and the user would only require to have one of these browsers. Implementing a SMIL player instead of using an embeddable SMIL player would only increase the time spent on a field that is not important for this research. Therefore a SMIL player embedded in the browser is used as the view of the compositions.

The design of the controller component is explained next.

3.5 Controller

Since the view of the compositions uses a browser and an embedded player, it is a logical choice to use a web interface for the controller. The user requirements would otherwise increase which is in conflict with requirement 7. For the creation of the web interface the Google Web Toolkit (GWT) is used. GWT can 'quickly build and maintain complex yet highly performant JavaScript front-end applications in the Java programming language.'[5] It compiles Java code into several JavaScript files that are all optimized for a particular browser. A user only has to download the JavaScript that is needed for his browser. Using GWT conforms to requirement 5, because the system can be accessed through a browser over the Internet. The system remains scalable as well, because the JavaScript files can be hosted at the App Engine. GWT furthermore allows for sharing of classes between the client and server when both sides are programmed in Java. Requirement 8 which states that the system allows for multiple people editing the same composition is not excluded, because the controller can be accessed by multiple users at once.

3.6 How all components work together

Requirement 9 states that the created music compositions are persistent. The Google App Engine does not allow access to the file system, so the music compositions cannot be stored on disk. To store data between sessions the App Engine datastore can be used. This is a scalable object database. The model component stores the compositions and their entire version history in the App Engine datastore to prevent compositions from being ruined and lost completely. In such a situation the composition can be restored to a previous version, which conforms to requirement 10. When a user requests a composition through the controller component it gets the latest version of that composition out of the datastore. The user can then manipulate a local version of this composition by using the same controller. The composition and changes to the composition can be listened to by using the embedded SMIL player, which belongs to the view component. When the user is done editing and satisfied with the result it can commit the changes. The model stores these changes by adding the new version to the version history of the composition in the datastore. To conform to requirement 8, every user is informed on updates of the composition it is currently editing. If an update takes place that conflicts with the changes the user made on the local view of the composition, the user can choose to take the new version or keep its changes. What determines a conflict depends on the implementation of the system.

In the following chapter an implementation of the design is explained.

[2]Quicktime `http://www.apple.com/quicktime/`
[3]Realplayer `http://www.real.com/realplayer/`
[4]Ambulant `http://www.ambulantplayer.org/`
[5]Google Web Toolkit `http://code.google.com/intl/en/webtoolkit/overview.html`

CHAPTER 4

Implementation

In this chapter an implementation of the system designed in the previous chapter is discussed. The server side as well as the client side is programmed in Java using the Eclipse IDE. The program uses the Google Web Toolkit v1.6.4 and the Google App Engine v1.2.1 packages. In appendix A an overview of the classes in the implementation can be seen. Not all classes and relations are shown. Classes reside in either the client package or the server package. When the application is deployed, the entire client side package is compiled to JavaScript. Classes in the server package are run as Java bytecode on the Google App Engine. First the workings of the server side implementation is explained, followed by the client side implementation. We conclude the chapter with a discussion on why the implementation conforms to the design.

4.1 Server implementation

The server side is an implementation of the model component from the previous chapter. In order to conform to this design the server should store the compositions, allow multiple users to edit a composition at once and keep track of a version history of the compositions. Appendix B shows a detailed view of the server package.

4.1.1 Storing the compositions

The compositions are stored in the datastore in several pieces (`Composition`, `Track`, `Revision`). Because the compositions should be editable by multiple users at once a composition is not stored as one object. Only one user at a time can edit an object in the datastore. The compositions are therefore split into several tracks (`Track`). A user can add tracks to a composition, edit tracks in a composition or remove tracks from a composition. One user at a time can edit a track. Therefore storing the composition this way allows a number of users equal to the number of tracks to edit a composition at the same time. This number is higher if the users are only adding tracks or less if the users are editing the same track. Each track keeps track of all the revisions that the track has had (`Revision`). By not removing tracks that are no longer in use by the composition but only indicating that they are removed, the version history of the entire composition is kept. A revision consists of a `String` that represents zero or more audio media objects in the SMIL representation (see previous chapter). Tracks are not further divided into multiple sample objects. The splitting of tracks into multiple sample objects is left for a future implementation if this one proves successful. The current `Sample` class that can be found in Appendix B has no relations to the composition. It is only used as a way to keep track of several URLs where audio files can be found to provide an interface for the client side to choose between a variety of samples that can be used in the composition (`AddSamplePopup` in appendix C).

4.1.2 Accessing the compositions

The server side provides various services to access the data in the datastore (`CompositionServiceImpl`, `SampleServiceImpl`, `TrackServiceImpl`). They offer methods to retrieve, edit and store objects. The server side is the only one with access to the datastore. The client side communicates with the server side through asynchronous calls. The objects on the server side are changed to client side readable objects when data is sent between the two (`CompositionClient`, `SampleClient`, `TrackClient`). Each entity in the datastore can be written to about five times per second. When multiple servers try to update the same entity simultaneously, one of the updates fails due to the optimistic concurrency control of the datastore. The update is repeated several times before an error is returned to the client. A composition can be updated by multiple users at the same time, because the composition is split into several track entities. Each track entity can be updated parallel. The track entity itself cannot be edited by multiple users at the same time. When a track is updated the server checks whether or not the track has changed in the time from the moment the client requested the track and the time the client committed the track. If so, the update fails, if not, the track is updated.

In the following section the client implementation is discussed.

4.2 Client implementation

An overview of the client side is given in appendix C. The client side implements the controller and view components from the previous chapter. To conform to this design the client should be able to request a composition, edit the composition, listen to the composition and commit changes to the composition.

4.2.1 Overview

When a client accesses the system, various options to manipulate the model are presented. The client can start a new composition (`NewCompositionPage`), edit a composition (`EditCompositionPage`), browse the stored compositions (`BrowseCompositionsPage`) or add a sample to the database (`AddSamplePage`). The `EditCompositionPage` implements the design from the previous chapter. It is used by clients to edit a composition.

4.2.2 Editing the composition

The client requests the latest composition from the server through the `EditCompositionPage`. The composition consists of several tracks, which are all loaded in the `MultiTrackEditor`. The `MultiTrackEditor` creates an interface that allows users to manipulate a composition locally. Users can add tracks (`AddTrackPopup`), edit tracks (`TrackButton`, `EditTrackPopup`), add samples to tracks (`AddSamplePopup`) and edit samples (`SampleButton`, `EditSamplePopup`). The user can hear its changes by using the Realplayer (`Realplayer`). While the editing of the composition takes place the `EditCompositionPage` constantly polls whether one of the tracks of the composition has been updated. If so, the user can update the local composition with the new version of the tracks. When the user has changed a track that has been updated, the user can choose to take the new version or use his version. Committing works the same.

In the next chapter the results of testing the implementation are presented.

CHAPTER 5

Experiments

This chapter presents the results from testing the implementation from the previous chapter. First the system used for testing is described. Then a test of all requirements is presented.

5.1 The system used for testing

The implementation runs on the Google App Engine. When the application is hosted at Google App Engine, statistics can be viewed through the Google App Engine Admin Console (Figure 5.1). It is a tool that shows the resources your application has used, gives traffic analysis and shows the data in the datastore. The resources your application uses are free in the App Engine up until a certain limit, after that each individual use of an extra resource needs to be paid. Table 5.1 gives an overview of the quotas on the resources this application can use.[1] For the experiments only the free quotas were used.

An extra server is used to store a large amount of audio files that can be used in the compositions. In the experiment the sample pack 'Shock the Monkey' from Peter Gabriel was used.[2]

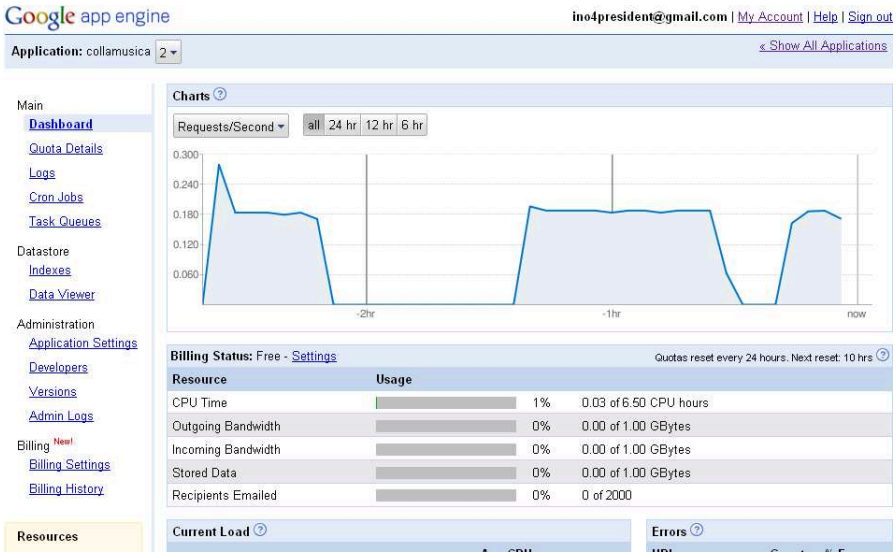


Figure 5.1: Google App Engine Dashboard

[1] http://code.google.com/intl/nl/appengine/docs/quotas.html
[2] The Real World Remixed `http://www.realworldremixed.com/`

Resources	Daily Quota Free	Daily Quota Billable	Maximum Rate Free	Maximum Rate Billable
Requests	1,300,000 requests	43,000,000 requests	7,400 requests /minute	30,000 requests /minute
Outgoing Brandwith	1 gigabyte	1,046 gigabytes maximum	56 megabytes /minute	740 megabytes /minute
Incoming Brandwith	1 gigabyte	1,046 gigabytes maximum	56 megabytes /minute	740 megabytes /minute
CPU Time	6.5 CPU-hours	1,729 CPU-hours maximum	15 CPU-minutes /minute	72 CPU-minutes /minute
Datastore API Calls	10,000,000 calls	140,000,000 calls	57,000 calls /minute	129,000 calls /minute
Stored Data	1 gigabyte	no maximum	None	None
Data Sent to API	12 gigabytes	72 gigabytes	68 megabytes /minute	153 megabytes /minute
Data Received from API	115 gigabytes	695 gigabytes	659 megabytes /minute	1,484 megabytes /minute
Datastore CPU Time	60 CPU-hours	1,200 CPU-hours	20 CPU-minutes /minute	50 CPU-minutes /minute

Table 5.1: Quotas of the Google App Engine

5.2 Testing the requirements

In order to determine if the system conforms to all requirements every requirement was tested individually.

1. *A music composition consists of multiple samples either placed parallel to each other on different tracks or in sequence with each other on the same track. Not required, but preferred: music compositions consisting of samples placed at any given point on a time line.*

   The application provides an interface to create compositions.

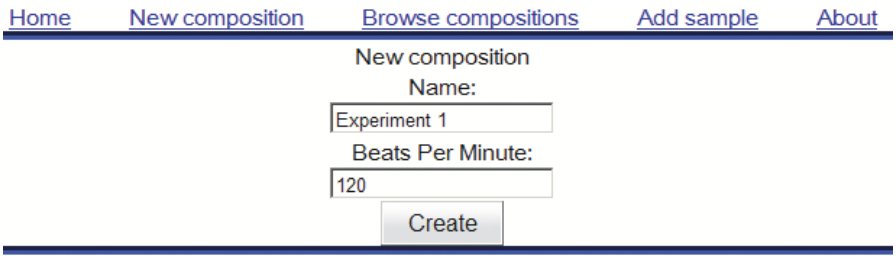


Figure 5.2: New composition

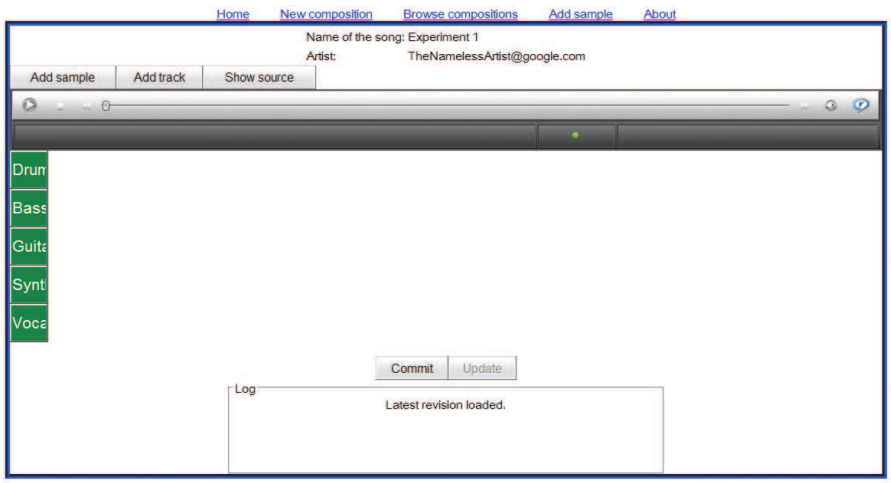


Figure 5.3: Empty new composition

When a composition is created, tracks and samples can be added to the composition. Samples can be inserted at a specified time in seconds. Once placed, the samples and tracks can be edited and removed as well.

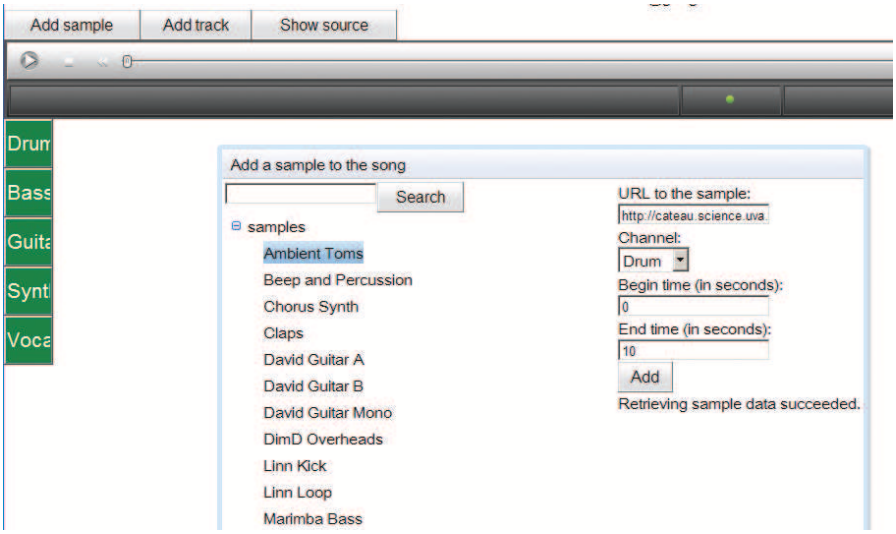


Figure 5.4: Adding a sample

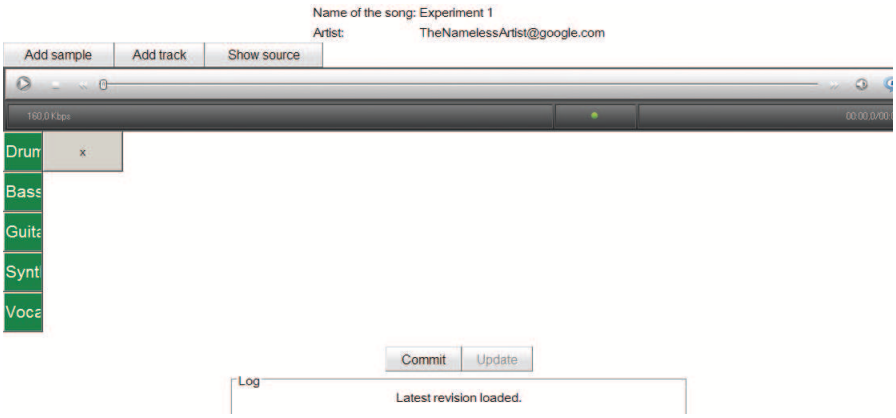


Figure 5.5: The sample has been added to the composition

The composition created in this experiment can be found at `http://3.latest.collamusica.appspot.com/#57073`. A movie that demonstrates the creation of this composition can be found at `http://3.latest.collamusica.appspot.com/#0`. To create a composition with the application go to `http://3.latest.collamusica.com/`.

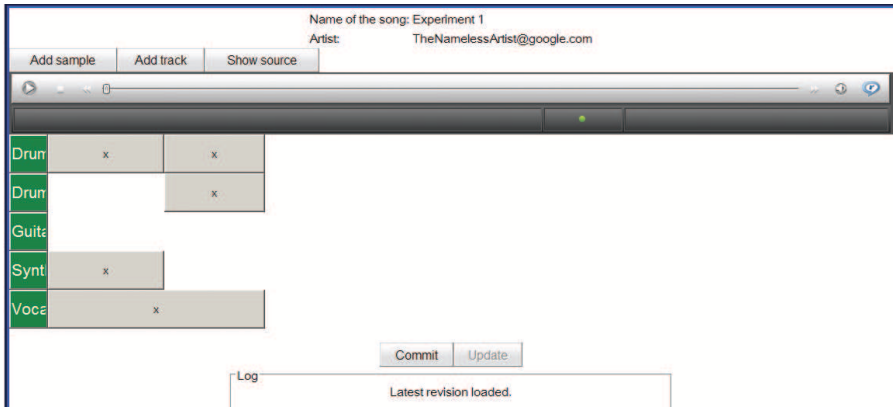


Figure 5.6: A composition with tracks and samples

2. *A sample is an audio file in either the Motion Picture Expert Layer Group-1 Layer 3 (MP3), Ogg Vorbis (Ogg) or Waveform audio format (WAV). These are popular formats that most users will be familiar with. This allows musical artists to create samples for the system using all popular recording software.*

   The use of SMIL does not restrict the use of a specific audio format, so whether this requirement is fulfilled depends on the player that is used to play the compositions. The application only allows playing the composition with Realplayer. Realplayer supports MP3 and WAV.

3. *The volume of every sample can be changed individually.*

   Although the SMIL specification allows this, Realplayer does not incorporate the part of the specification that allows the specification of volume levels. The requirement is not met.

4. *A music composition that has been created with the software is audible. The ability to play a composition outside of the system using standard players is a plus.*

   The compositions are indeed audible using Realplayer in the application. Outside of the application the compositions can be listened to using Ambulant, Quicktime and Realplayer.

5. *The system provides network access over the Internet to support access from geographically distributed locations.*

   The system is accessible over the Internet using a browser. This requirement is met.

6. *The system is scalable. If the system becomes popular it should be able to deal with the increase in user numbers.*

   The number of requests the application can handle are given in Table 5.1. The App Engine distributes the application over multiple CPUs when necessary. The request rate per user is on average one request per 4 seconds when editing a composition. This is equal to 15 requests per minute, which based on the 7,400 requests per minute quota allows for nearly 500 users editing a composition at the same time. The CPU time is 34 ms per request. 34 ms $\times$ 15 requests per minute = 0.51 seconds. Based on the 15 CPU minutes per minute quota, the number of users the system can handle on average is more than 1700. The

bottleneck in this case thus is the number of requests the system can handle per minute. The requirement is met, the system allows for nearly 500 users editing a composition at the same time.

7. *The requirements for a user to use the system should be low, so that it reaches a broad audience.*

   Only a browser and Realplayer are required. However, the combination of Realplayer and SMIL does not work very well in Internet Explorer[3] and Safari[4]. The requirements for a user to use the system therefore becomes Realplayer and either Firefox[5] or Chrome[6] in practice.

8. *The system can deal with multiple people editing the same music composition.*

   When editing a composition, the application checks every 4 seconds if an update of the tracks of the composition has taken place. If so, the user can choose to update or not. If an update cannot be done, because the user edited the track, the user can choose to force an update or keep his version. When a user commits its changes, but the commit fails because one of the tracks was edited in the time the user received the track and the time the user committed the track, it can choose to force update the track. In figure 5.7 various scenarios that can occur are shown. User 1 has committed a new version of the 'Drum' track, the commit succeeded and he updated his tracks afterwards. The update in this case was the update of the 'Drum' track which he committed himself, so merging the new track with the tracks he had caused no conflict. User 2 on the other hand changed the 'Drum' track as well, but his version is in conflict with the new version. He can either choose to keep his version or take the new one. User 3 had the same experience and chose to keep his version. Now he tries to commit his version, but the track that he originally received is older than the one now stored in the datastore. To overwrite this new version the user can choose to force commit.

[3]Internet Explorer `http://www.microsoft.com/windows/internet-explorer/default.aspx`
[4]Safari `http://www.apple.com/safari/`
[5]Firefox `www.mozilla.com/firefox/`
[6]Chrome `http://www.google.com/chrome`

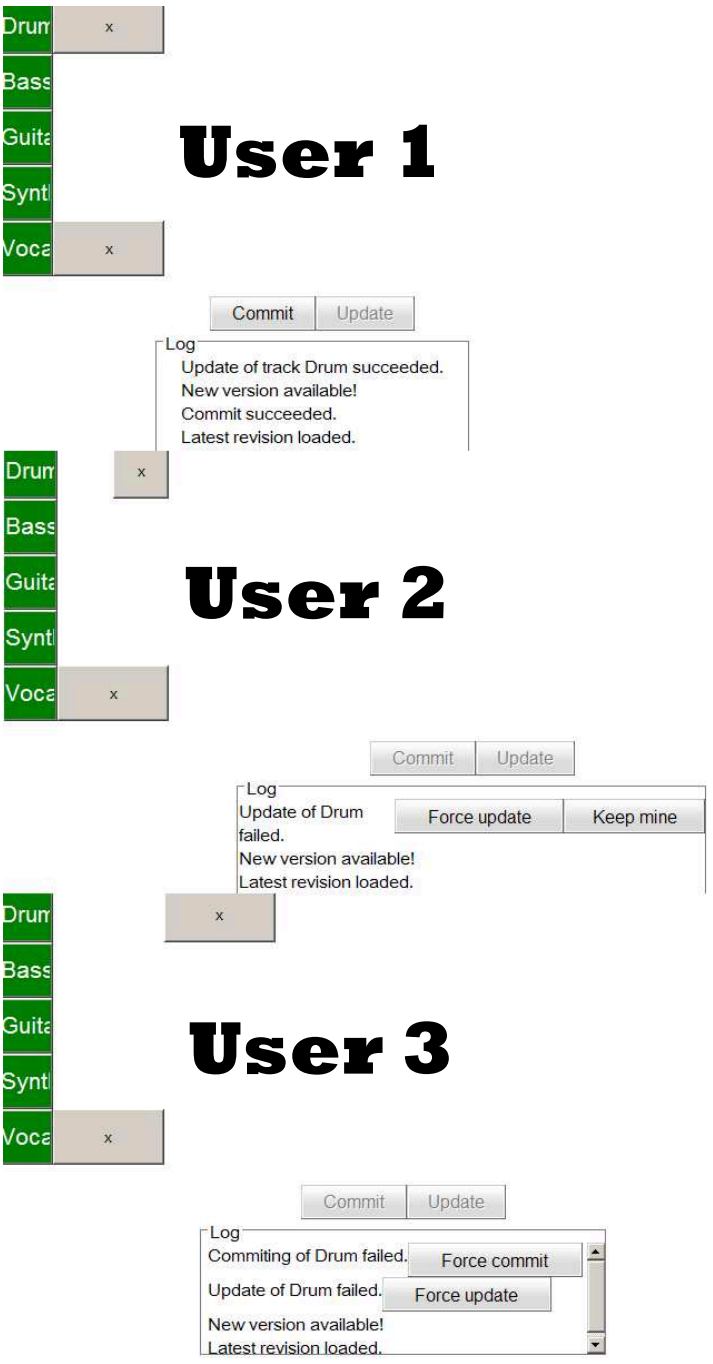


Figure 5.7: Various scenarios in a multiple user session

9. *The music compositions are persistent. This way the music compositions can be edited from distributed locations in time.*

   The music compositions are indeed persistent. When music compositions are added to the datastore they can be edited at any time.

10. *The provenance of a music composition is saved. In case the music composition is changed in an undesirable way, the music composition can be restored to a previous version.*

    All revisions of the tracks are stored. They can be viewed in the Google App Engine Admin Console. The music composition can also be restored to a previous version by using the Admin Console.

5.3 Feedback

During the testing several users gave feedback. Here is list of suggestions by the testers:

- Show a time scale. The multi track editor indicates the length of all samples on the tracks by drawing them with a width proportional to their length on the screen. Because no time scale is shown it is hard to say what the length of the sample really is.
- Dragging and dropping of samples. Adding and editing samples works, but it would be easier if the samples could be dragged to the desired location and size.
- Add samples at a certain measure or bar instead of just at a specified time in seconds. Dividing a composition in measures and specifying samples to be played from a certain bar until a certain other bar is a popular technique for creating music compositions.
- The progress indicator on the Realplayer should also be shown on the multi track editor. When the composition is played, it is not clear what samples are played at what time.
- Store and show the effects instantly instead of having to commit and update. The interface for committing and updating is not very comprehensible for must users. Direct manipulation of the model on the server is more understandable and requires less interaction by the user to function.

CHAPTER 6

Conclusions

In this thesis an alternative solution to the problem of performing music composition in a distributed collaborative environment has been presented. The solution is unique in several ways. Most notable features are the use of SMIL for the representation of the compositions and the use of Google App Engine to create a scalable system. Also the placement of samples on a time line and the ability to edit a composition with multiple artists at once are new ideas that cannot be found in other solutions. The biggest flaw in the implementation is the use of Realplayer to play the compositions. One problem is that the player only plays the SMIL compositions properly in Firefox and Chrome, the other is that it does not follow the SMIL specification completely. Because of this the volume of samples cannot be altered, which removes all sound mixing features from the software. A different problem in the implementation is the user interface. It is too complicated for average users and needs to be improved. This would allow more users to utilize the system and only then the success of the solution can be determined. Until then the research did give new insights on music composition in a distributed collaborative environment, but did not provide an utilizable solution.

6.1 Future Work

The concept of committing and updating works less intuitive then expected. A different solution to multiple artists editing the same composition should be researched. Google will soon release a tool that could improve the current implementation. The tool is called Google Wave.[1] Google Wave allows text to be edited collaboratively and have changes be seen to everyone else that's viewing the document almost instantly. The tool could be used for the SMIL compositions on the App Engine to allow multiple users to edit a composition at once and have every user see each others changes immediately. This might work more intuitive and dynamic than a version management system. The feedback given by the users during the experiments is a valuable asset for further development of the system.

[1] Google Wave `http://wave.google.com/`

Bibliography

[1] Jörg Stelkens *peerSynth: A P2P Multi-User Software Synthesizer with new techniques for integrating latency in real time collaboration* 2003.

[2] Nicolas Bouillot *The auditory consistency in distributed music performance: a conductor based synchronization* 2004.

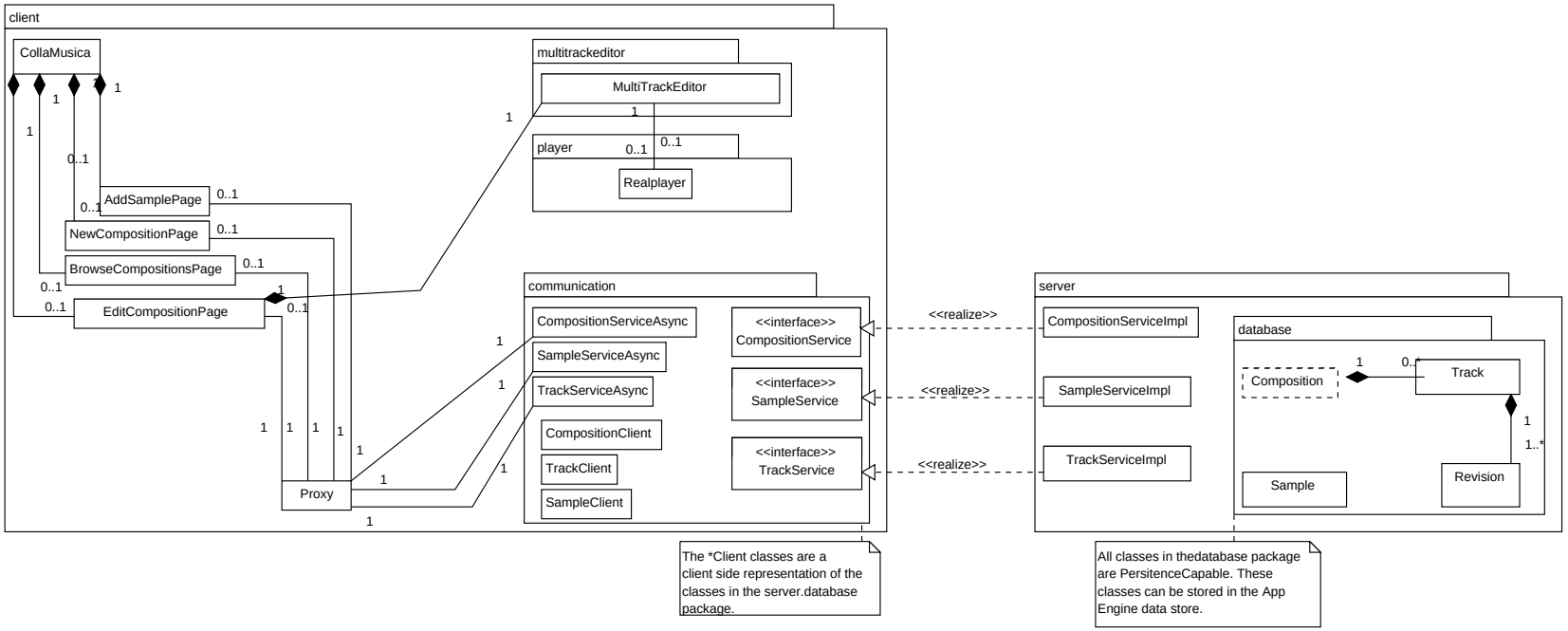
[3] Craig Latta *Notes from the NetJam Project* 1991.

[4] J. Ox and D. Britton *GridJam* Creativity and Cognition: Proceedings of the 5th conference 2005.

[5] Synchronized Multimedia Integration Language (SMIL 3.0) `http://www.w3.org/TR/SMIL/`

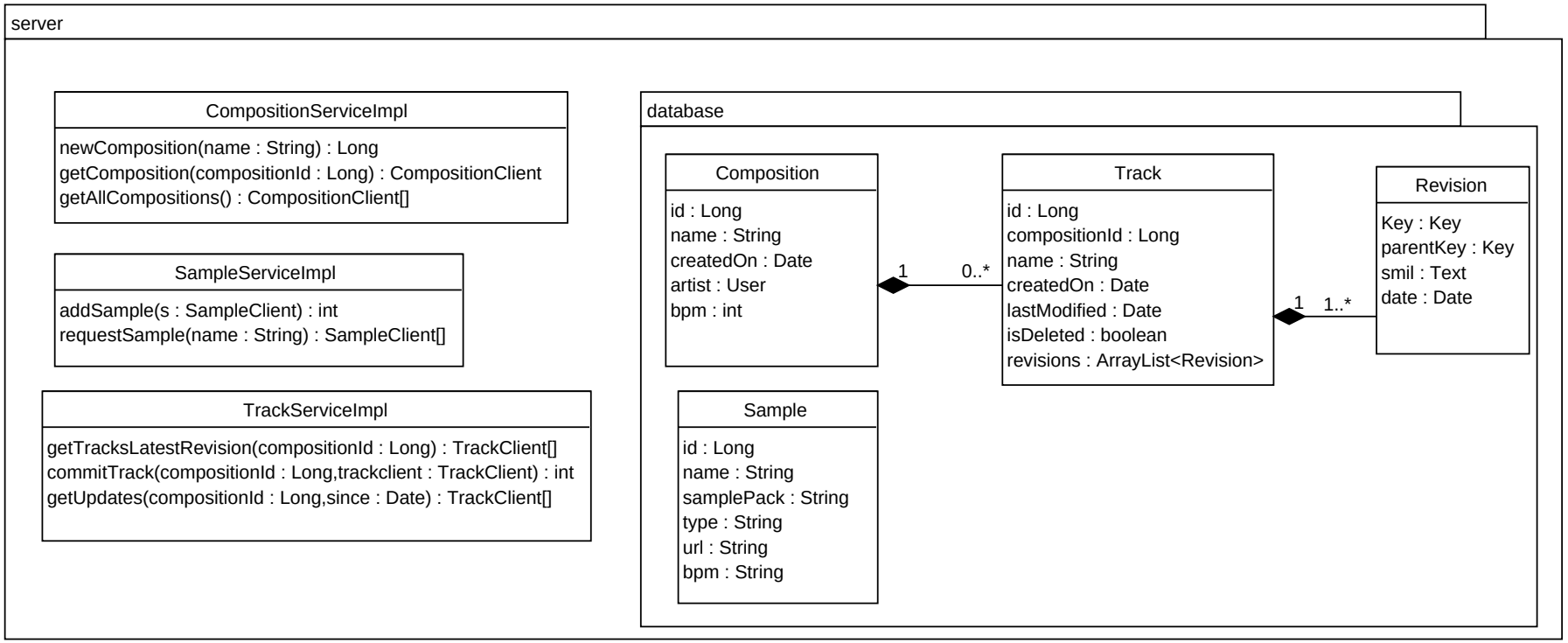
Appendix

Appendix A



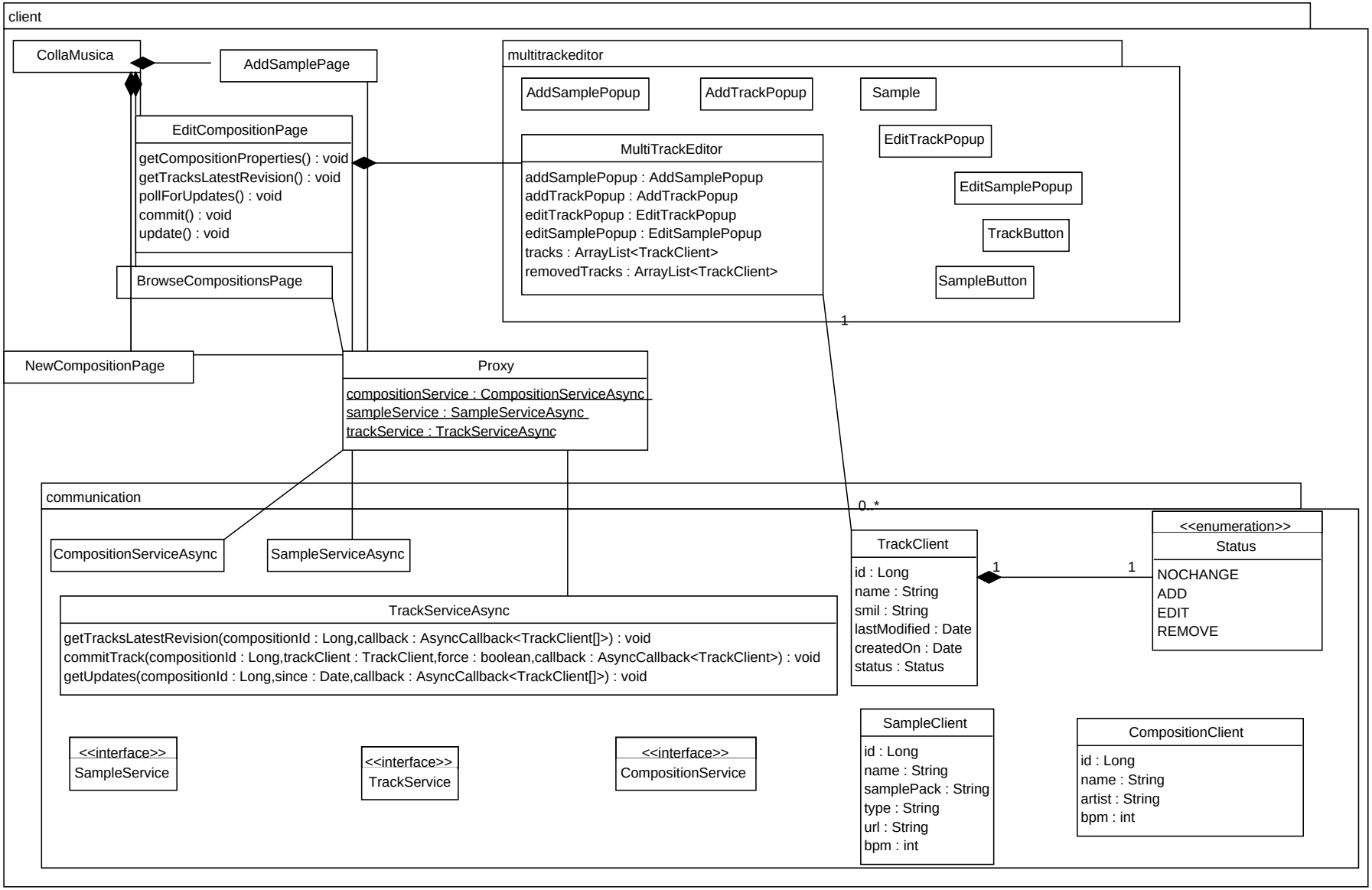
Implementation overview

Appendix B



Server implementation overview

Appendix C



Client implementation overview