Design and Implementation of a Collaborative Music Software

Master's Thesis

Johannes Schultz
Master Informatik

Betreuer:
apl. Prof. Dr. Achim Ebert,
M.Sc. Franca-Alexandra Rupprecht

Abstract

Collaborative software, or groupware, allows users around the world to work towards a common goal without the need for explicit synchronization e.g. by locking shared files or sending updates via e-mail. Collaboration helps to bring together competences and ideas from different experts. It is a common tool in the industrial context, but also in music creation where different musicians can contribute to the composition of songs by working on different parts depending on their individual strengths and skills. In the context of music software, there are several approaches to allow collaboration between musicians, but no current approach exists for so-called *tracker* software, which are keyboard-driven expert systems.

This thesis seeks to address this issue by defining desirable features for a collaborative tracker and by extending an existing tracker software with a collaborative editing mode. The technical approach is generalized to serve as a guide for extending similar single-user software.

Zusammenfassung

Kollaborative Software ("Groupware") erlaubt es Benutzern auf der gesamten Welt, an einem gemeinsamen Ziel zu arbeiten, ohne dass dafür eine explizite Synchronisation wie z.B. durch Sperren oder den Austausch von E-Mails notwendig ist. Kollaboration hilft dabei, die Kompetenzen und Ideen von verschiedenen Experten zu vereinen. Kollaborationswerkzeuge sind insbesondere im industriellen Kontext verbreitet, aber auch die Erstellung von Musik kann davon profitieren, wenn verschiedene Musiker zu einer Komposition beitragen, indem sie an verschiedenen Teilbereichen abhängig von ihren Fähigkeiten und Stärken arbeiten können. Im Kontext von Musiksoftware gibt es verschiedene Ansätze, um eine Kollaboration zu ermöglichen, aber es gibt keine aktuellen Ansätze dafür im Bereich der *Tracker*-Software, welche tastaturgesteuerte Expertensysteme darstellt.

Diese Arbeit befasst sich mit dem Problem, indem wünschenswerte Eigenschaften für einen kollaborativen Tracker definiert werden und ein existierender Tracker um einen Kollaborationsmodus ergänzt wird. Der technische Ansatz wird verallgemeinert, um als Leitfaden für die Erweiterung ähnlicher Einzelbenutzersoftware dienen zu können.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context and Motivation

Writing and performing music is an inherently collaborative task, as the multitude of bands and orchestras in this world show us. Starting with early sequencing programs in the 1980s such as *Steinberg Pro 16* [22], digital music composition tools have evolved significantly and are now the prevalent way of composing music.

However, most mainstream music composition software either does not offer any collaboration features to work on a piece of music with another musician on a different computer, or the support of collaborative features is only very limited. An informal survey among eight fellow musicians has shown that many artists do not use these collaborative features because they are either not real-time or simply buggy.

While many electronic musicians seem to be comfortable with this way of working, it is inherently limiting to those who wish to collaborate with one or more fellow musicians. Just like the various instrument parts in a classically notated piece of music are often written by the individual performers of each instrument in a band, the same should be possible for electronic compositions. For example, one musician may be more comfortable writing bass lines, while another one might prefer writing leads and background chords, and a third one could add the drum section.

There exists a variety of tools for writing music with a computer; The majority of them can be classified as **sequencers**, but there is also a small group of music software called **trackers** which have a steeper learning curve compared to more mainstream software, but are suited well for realizing musical ideas very quickly using only a computer keyboard. Trackers revolve around a text grid called **pattern**, in which notes and other control data are entered. When interpreting these patterns, the computer can then send the note data to a variety of sound sources, from basic samples (audio recordings) to complex virtual instrument plugins. This textual representation and keyboard focus make trackers very efficient to work with but also more difficult to learn, much like expert systems in other domains. There are no current collaborative tools available for this niche, leading to work-arounds like exchanging song project files via e-mail.

1.2 Goals

The goal of this thesis is to enable tracker musicians to make use of collaborative features in an existing tracker software by extending it with online editing features. Collaboration should not be limited to a handful of actions but all the program's functionality should be available to the musician. Furthermore, the steps taken to enable this goal should be analyzed and generalized to other single-user software, hopefully making it easier for other software developers in the future to add collaborative features to their software. The results will be quantified in order to determine the usability improvement achieved by the collaborative features.

1.3 Outline

In the remainder of this thesis, we are going to explore related work in the field of collaborative and music software in Section 2, the methodology for creating a collaborative tracker software in Section 3, the technical choices that have been made in Section 4.1, the actual implementation in Section 4.2 and a user study in Section 5, finally leading to a discussion and conclusion in Section 6.

2 Related Work

In this chapter, we are going to explore and define collaborative software in general, music software and collaborative music software.

2.1 Collaborative Software

While many software systems (including music software) only support single-user interaction with the system, collaborative software, or *groupware*, allows asynchronous group activities to be carried out. Asynchronous operations allow to simultaneously work on and modify the same objects and see each other's modifications in real time. This means that a group of people can work on a **common task**, no matter if they are in a **shared environment** or physically apart. Groupware can be used in many contexts. The scientific Computer Supported Cooperative Work (CSCW) community has explored use cases such as outline and graph editing [13], in the medical field [4], orthography systems [8], software engineering [27], landscape planning [47] and factory layout planning [39]. Probably the most popular and well-known groupware these days are office suites like *Google Docs* or collaborative text editors like *Etherpad* [46].

Ellis et al. [13] define two dimensions over which collaborative systems can be classified: the common task dimension describing how much related the tasks of individual users of the system are, and the shared environment dimension describing how close the participants are physically. Groupware ranks high in the common task dimension (as opposed to e.g. a time-sharing system), because all users work towards a common goal. Groupware can rank both high and low in the shared environment dimension depending on the usage scenario. The intent of this thesis is to develop a system which ranks high in the *common task* dimension (as several musicians will work on the same piece of music with the same intents), but low in the *shared environment* dimension, as the software must support collaboration with remote musicians.

Real-time groupware systems have the following characteristics according to Ellis et al. [12]: highly interactive, distributed (users may be in different physical locations), volatile (participants can join and leave), ad hoc (participant actions do not follow a planned script) and focused (participants work on the same data, causing a high number of access conflicts). As a result, groupware poses unique challenges such as maintaining consistency of the shared documents between all collaborators, while also offering a short response time when propagating the actions of a user to all other users, and concurrent editing [43]. Groupware is different compared to other multi-user software such as database systems that try to give the impression that there is only one user by means of locks and

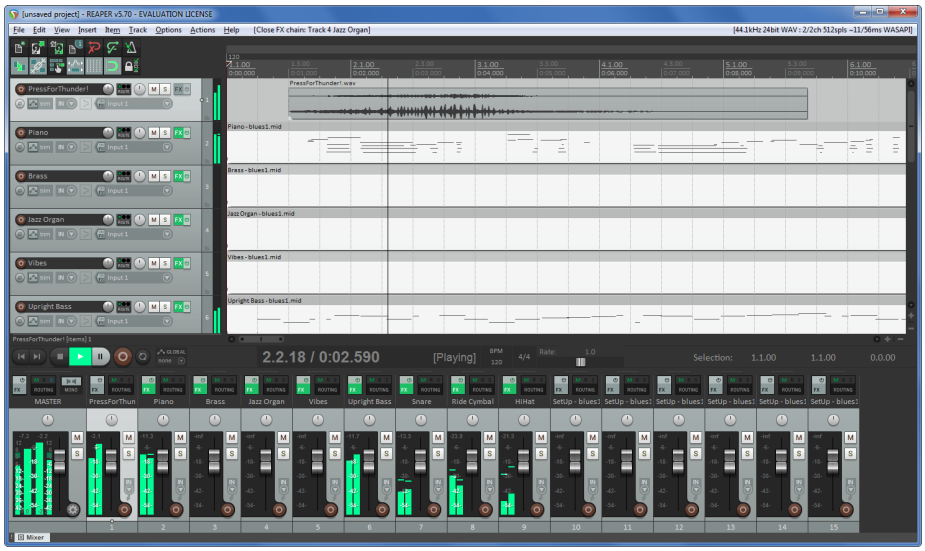

Figure 2.1: *REAPER*, a typical sequencer program.

transactions: groupware wants to achieve the opposite impression and requires concurrency control to resolve conflicts between users.

2.2 Music Software

The most common form of music composition program on computers is the **sequencer** [32], a tool that revolves around a horizontal linear timeline showing the *arrangement* of the bigger building blocks of a music project, such as streams of Musical Instrument Digital Interface (MIDI, a digital communication protocol for exchanging note and modulation data between instruments and computers) events, automation envelopes and audio tracks. These building blocks can be freely arranged. Simple editing of the blocks is also possible, but the majority of lower-level editing work is relayed to a **piano roll** or **score editor**, where the individual note events of each MIDI track can be edited with high precision. Most interaction is done through dedicated MIDI controller or the computer mouse, although keyboard shortcuts can be used to speed up some actions [32]. Popular examples of sequencers [32] include *Steinberg Cubase* [16], *Ableton Live* [1], *Tracktion* [10], *REAPER* [25] (Figure 2.1) and *FL Studio* [23].

On the other hand, **trackers** are centered around a concise textual notation arranged in fixed grid (**pattern**) much akin to a spreadsheet. A pattern contains a number of simultaneously playing **channels** that contain note data, instrument information and modulation effects. Time normally runs from top to bottom in a pattern, while channels are arranged from left to right. A song is typically made up from multiple patterns and a set of instruments, which can be based on sample data, external MIDI devices or virtual instrument plugins. Patterns are visualized as a cell grid, as seen in Figure 2.2. Instruments are triggered in patterns by specifying which note to play, together with some op-

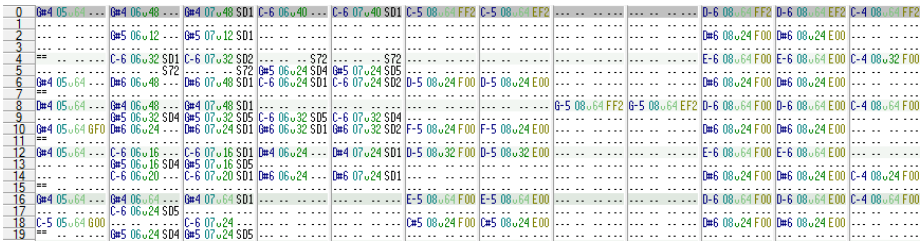

Figure 2.2: Example of some musical note data in *OpenMPT*, a tracker program.

tional effects which modulate the playback of the note. For example, the cell content `D#6 01 v32 F01` may instruct the program to play the note `D#6` using the first instrument (`01`) at 50% velocity (`v32`), slowly increasing its pitch over the course of the current row (`F01`) [33].

Trackers originated on the *Commodore Amiga* home computer, where they made optimal use of the existing sound hardware to play background music with up to four simultaneous voices (instruments) in games and graphic demos with minimal computational overhead [38]. Later, the tracker concept was expanded, in particular on the *Personal Computer* (PC), to allow for more polyphony, wider note ranges, bigger patterns, more (and more expressive) instruments and generally making trackers more intuitive and easy to use, eventually turning them into general-purpose music composition tools rather than specialized tools for game developers.

In direct comparison, the main view of a sequencer focuses more on the "big picture" (the arrangement), having the detailed note information hidden in lower-level views, while the main view in a tracker dedicates most of the screen estate to all the note data, offering many editing actions to quickly transform the notes. A lot of properties of sequencers are modeled after analog audio gear, such as the playback controls (transport bar) and the mixer [32].

Figure 2.3 shows the general structure of a song (also called **module**) as it is found in many trackers:

- A set of global settings describes general properties of the song such as its title, the initial tempo, initial volume levels and similar properties.
- Audio plugins, which are either effects (such as reverberation or filter) or virtual instruments.
- Samples, which consist of the actual sample data (waveform) and settings such as pitch, loop points, sample name, volume, etc.
- Instruments, which describe how samples or instrument plugins are supposed to be played. An instrument references a number of samples or a plugin.
- Patterns, which contain the score of the music.
- Order lists, which are sequences of patterns that describe in which order the patterns are played.

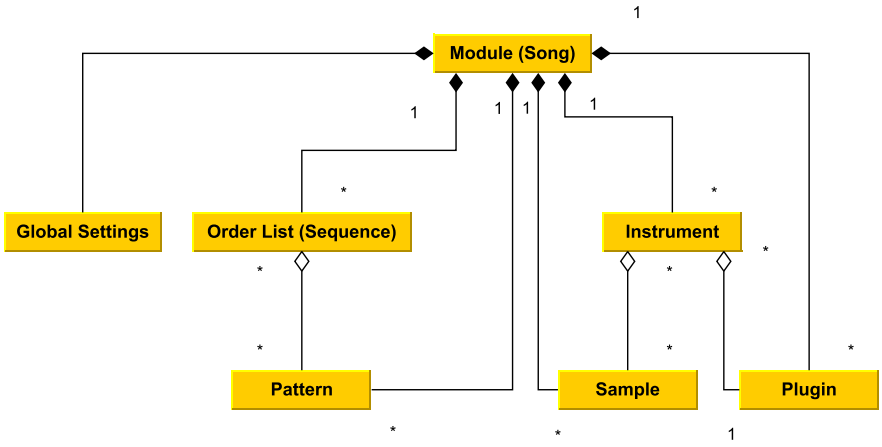

Figure 2.3: Structure of a "module" (song) as used in many trackers.

Popular examples of current tracker software include *Buzz* [44], *MilkyTracker* [3], *OpenMPT* [41], *Renoise* [30] and *Schism Tracker* [45].

In their research, Nash et al. [33] [34] highlight that the tracker interface allows for rapid editing interaction with a fast edit-audition feedback cycle, which offers a high degree of liveness, enabling sketching and flow, but at the same time there typically is a steep learning curve, because motor skills and program knowledge have to be learned and practiced much more in-depth than in more graphically-oriented sequencer software. This learning curve can make trackers cumbersome to use initially, because a lot of functionality is hidden behind shortcuts that have to be figured out first. Once the user is familiar with the user interface, however, a tracker allows many tasks to be carried out very quickly and efficiently.

As a result, it can be said that trackers are more complicated **expert systems**. Expert systems are based on knowledge rather than information and are "used primarily by the people who built them, love them and are tolerant of their idiosyncrasies." [6] Trackers are suitable for expert users, but little has been done in the past to enable this advantage in liveness and flow for collaborative musicians. Among all the widespread tracker software released in the last thirty years, only the MS-DOS-based *Impulse Tracker* [28] stands out by having a collaborative editing mode via the now-obsolete IPX network protocol. However, this collaboration mode was only added as an afterthought to the software in 1999, four years after its initial release, and many editing features were disabled in collaborative mode. Thus, most tracker musicians have to resort to the "classic" way of collaboration by means of exchanging music files and taking turns at editing them, and the immediacy of collaboration is not available to them.

2.3 Collaborative Music Software

The idea of collaborative music software is not new: there is a number of existing music software that enables collaboration, and also add-ons that can enable collaborative features in single-user environments to some extent. However, a lot of this software comes with various disadvantages that this thesis seeks to address:

- Dedicated sequencers like *Ohm Studio* [37] require learning a new interface, rather than being able to work in a familiar environment the musicians are already used to.
- Some solutions are hosted in the "cloud" like *Kompoz* [26], which can raise privacy and reliability concerns. Once the service shuts down, it may be impossible to retrieve the song data and reuse it in a different music tool.
- A lot of this software is relatively new compared to single-user solutions that have been around for 20 or 30 years, thus it may lack features or may not be as mature.
- Some solutions cannot be used for a full song production, because they only focus on sharing the musical notation data, but not instrument definitions (e.g. *MuseScore* [31] or *flat.io*). The ability to use custom instruments is crucial to electronic music production.
- On the other hand, solutions like *Kompoz* [26] only appear to allow collaboration with prerecorded stems. This means that it is not possible to edit another collaborator's song parts, as e.g. an entire melody or chord sequence is "baked" into a single monolithic audio stream called *stem*. Only the artist who created the stem can also update it.
- There is also a completely different branch of collaborative music software used for jamming, such as *NINJAM* [24]. These programs do not allow for complete productions and have fundamentally different requirements such as low latency transmission and perfect audio synchronization between clients, while lacking editing features.
- Add-on solutions like *Blend* [5] extend existing music software. They are inherently limited by their design and cannot offer real-time collaboration, as they just offer automatic synchronization of project settings, audio files and other related data.
- Solutions like *Steinberg*'s *VST Connect Pro* [17] for *Steinberg Cubase* have often been reported as being very unstable.

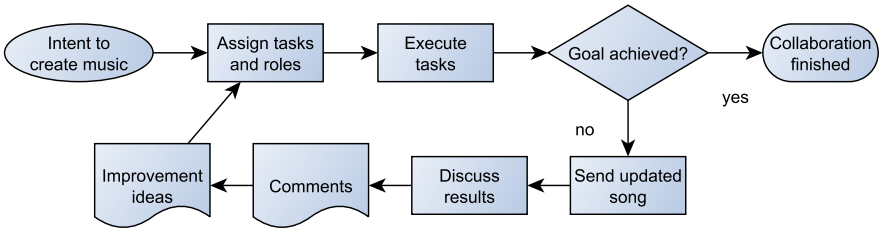

Figure 3.1: General workflow of music collaboration

3 Methodology

In this chapter, the methodology used to implement a collaborative tracker software is explained and motivated. An existing music composition tool is chosen and enhanced with collaborative features, thus improving the workflow of joint music composition by making it more efficient and feature-rich. A number of collaboration-supporting features is chosen and rated according to the criteria catalog defined by Rupprecht et al. [40].

3.1 Collaboration Process of Music Composition

To better understand the requirements for collaborative composing, the typical collaboration process needs to be examined first.

Collaboration can happen in many different ways, so editing of all the components described in Figure 2.3 needs to be supported. A song typically grows dynamically during its creation, and tasks during the songwriting process include:

- adding instruments,
- adding notes and patterns,
- adding plugins,
- revising old ideas,
- but also throwing away earlier sketches.

As a result, the ability to insert, modify and delete new entities needs to be available at any time. Coordination between the collaborators is crucial, as they need to discuss the structure of the song and the changes they make. For a disciplined cooperation, it can help to restrict the editing of certain entities not only verbally, but also apply technical restrictions so that users do not get into each other's way.

After each collaborator's turn, the updated document has to be sent to the other musicians, e.g. through e-mail. The results can then be discussed, resulting in improvement ideas, and the next person can start editing. Everyone else is not allowed to edit the file at this point.

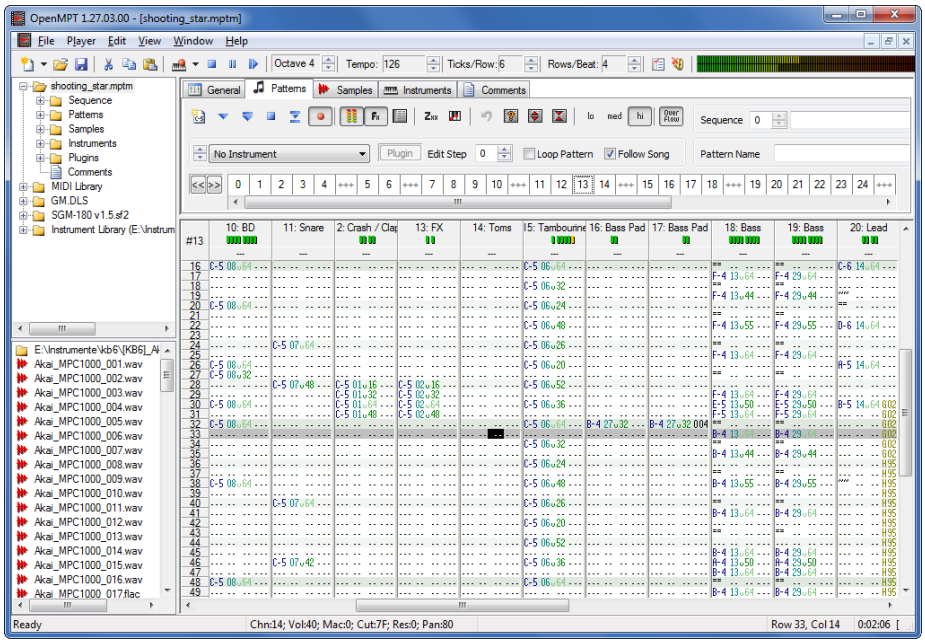

Figure 3.2: *OpenMPT*, a free tracker software.

Music composition can have a very dynamic nature with constantly changing roles and tasks, in particular when one musician gets inspired by the ideas of another. For example, one musician might get stuck in the middle of writing a melody, but a collaboration partner could get inspired by the half-finished melody and be able to find a continuation. The creation of a song can span several days, weeks or even months, so collaborators need to be able to leave permanent comments and write to-do lists.

The workflow described in Figure 3.1 roughly matches this very lively but sometimes also chaotic procedure.

3.2 Base System Design

As the base for the implementation of this thesis, the free tracker software **OpenMPT** [41] (Figure 3.2) has been chosen for several reasons:

- The need for implementing a full musical application from scratch was avoided.
- As trackers are expert systems with a steep learning curve, basing the collaborative environment on an existing, known environment avoids the need to re-learn a new software for many users, as they can use the same familiar tool for creating collaborative works as they previously used for other pieces of music.
- *OpenMPT* is one of the most popular trackers. Of all the tracked music works uploaded to *The Mod Archive* [42] in 2017, one of the largest col-

lections of tracked music online, it was found that about 42% are written in *OpenMPT*.

- After being developed for more than twenty years, *OpenMPT*'s code base is very mature and stable. I am one of the main developers and already familiar with the code.

In the past, I have worked together remotely with various other tracker musicians from various parts of the world, using *OpenMPT* and other tracker software. The workflow was typically as described in Section 3.1: One artist would start by picking a set of instrument files to be used for the song and writing the first few melodies, chords or whatever came to their mind. The resulting file would then be saved and sent via e-mail to the next musician who would add more instruments and musical parts. This process was repeated several times until the piece of music was done. In this workflow, ideas can be exchanged via e-mail or instant messengers between each iteration, but is difficult to directly discuss ideas by trying them out and listening to them together, and it is impossible to work in parallel on unrelated parts of the song. As a result, the collaborative functionality was spread over several components:

- Editing, insertion, deletion of data: Music software
- Data exchange: External file transfer
- Task and role assignment: External communication channel
- Comments: Music software (limited) or external communication channel
- Discussion: External chat software

Hence, a number of collaborative features to improve this situation had to be found, which are described in Section 3.3.

3.3 Applied Collaboration Methodology

Collaborative software can be categorized by the way the users progress towards a common goal. Depending on the approach, the location of users and division of labor change the way the software is used. In the case of this thesis, **distributed cooperation** as well as **distributed single task performance** are valid working styles for the collaborators, depending on whether they want to work on the same or different instrument or song section at the same time.

Collaborative features can be classified by the following groups according to Rupprecht et al. [40]:

1. **Content support:** Active interaction and integration of the content by the actors.
2. **Information sharing:** Functionality used to share information in order to establish the knowledge basis for all actors.
3. **Coordination support:** Tasks that support the coordination of work packages between actors.

4. **Communication support:** Features that facilitate communication to bridge spatial gaps.

5. **Compliance support:** Aids for fulfilling rules or guidelines.

6. **Content management:** Access control, data synchronization and consistency.

7. **Usability:** Methods to improve satisfaction, efficiency and effectiveness.

8. **User Experience:** Emotions and attitudes about using the technology.

Together with expert users of the tracker software that was to be modified, as well as some users from the informal survey mentioned in the introduction, a list of features that should be available for the collaboration was devised. In fact, there exists a feature request dating back as far as 2010 [36] for a collaborative network mode which has been used to come up with the initial requirements, which were then further refined. The requirements were then finalized in a task model as defined by [40].

1. Editing: Editing a song shared between collaborators should not be any different from editing a song in a single-user scenario. All single-user features should be available so that the user can work on the song as they are used to. In this context, editing comprises all possible user actions including creating, modifying and deleting content. This satisfies categories 1 (Content support) and 8 (User experience: Intuitive and simple, reduced cognitive load).

2. Chat: As one of the most basic collaborative features, the application should allow direct communication between all collaborators without the need of installing any external chat software. This feature supports coordination (3: jurisdiction) and communication (4: communication, discussion) by enabling users to discuss changes, task assignments and review each other's progress.

3. Rights management: The user that initially shares a song for collaboration decides how many collaborators can join the editing process. It should be possible to allow or deny certain actions for some users. As an example for this, two access levels should be implemented: Collaborator and spectator. Collaborators have full access to the shared document and can edit anything, while spectators can only watch the collaborators' actions and write chat messages. The latter can be interesting as a replacement for video live-streaming the creation of a song, which is done by both professional and amateur musicians on platforms such as Twitch these days. A song can be optionally protected by a password. This design supports category 3 (coordination support, in particular jurisdiction) and 6 (content management: access control).

4. Shared edit cursors: Collaborators should be able to see other collaborators' edit cursors in order to be able to synchronize their work and avoid collisions when editing pattern data. Since most of the time in a music project is spent working on the pattern data, the pattern editor is the most critical part of the user interface where such collisions have to be avoided. Similar edit markers could be added for samples and instruments as well. The edit markers should be shown both in the order list, giving a rough idea which pattern the collaboration partner is currently editing, as well inside the pattern, highlighting the exact cell the user resides in. Collisions can be avoided effectively

in this way. To tell different users apart, each user's edit marker is shown in a different color. This feature supports information sharing (2: Tracking of other users' approach, retrieval of context-relevant information, screen sharing, shared workspace), as changes by other users can be tracked easily, and implicitly also communication support (4) as the user's edit position is communicated automatically. Later, this feature can be extended to also show edit cursors for instruments and samples, where collisions are less likely to happen.

5. Automatically following edit cursors: To be able to better understand the actions of the collaborators with edit rights, users with spectator rights can choose to automatically follow the edit cursor of a specified user. This is similar to the *telepointer* concept described by Ellis et al. [13], but with the difference that the cursor can only be moved by one person, while still moving on other users' screens as well. This feature satisfies category 2 (information sharing) for the same reasons as the shared edit cursors.

6. Edit locks: To take collision avoidance even further, edit locks can be used to restrict the editing of a specific item to a single user. This way, it can be guaranteed that e.g. no data in a pattern is overwritten by other users. This feature facilitates coordination support (category 3: jurisdiction) and compliance support (5: team self-management).

7. Annotations: To enhance context-dependent communication, annotations can be added to discuss song patterns. Unlike chat messages, they are permanent (not lost between edit sessions). The goal is to help artists organize their collaboration spanning multiple sessions without having to write down thoughts they might be having about a specific pattern location in a different document. Like the chat feature, this supports coordination (3: alert mechanisms, awareness support) and communication (4: discussion tool).

Apart from functional requirements, the instability of some existing collaboration solutions mentioned before motivates to add quality aspects as non-functional requirements as well: the collaborative editing mode should not lead to random crashes and make *OpenMPT* less stable, in particular if the internet connectivity is not perfect.

4 Implementation

4.1 Software Architecture

The technology for realizing the goal of this thesis was chosen as follows.

4.1.1 OpenMPT

OpenMPT is written in C++14 and runs on the *Microsoft Windows* platform. The GUI is realized using the *MFC* framework, which is a lightweight toolkit based on the *WinAPI* – and the reason why the code only runs on *Microsoft Windows*.

4.1.2 Networking Components

For implementing the networking component, it was initially planned to use the widespread **gRPC** [18] framework. *gRPC* is a high-performance Remote Procedure Call (RPC) framework by *Google* that allows to define software interfaces for distributed processes on a variety of platforms using *Google Protocol Buffers* as the interface definition language. In theory, this sounded very promising, as the interfaces would only have to be defined in a simple, text-based definition file, from which the interface code would be generated automatically, saving a lot of time. However, *gRPC* is a huge project with a lot of dependencies. While it is easy to set up on UNIX-based systems, configuring and setting up all dependencies, as well as integrating the *CMake*-based build system into *OpenMPT*'s own build system proved to be very difficult and time-consuming, so in the end it was decided to look into different libraries instead.

After the attempt of integrating *gRPC* was unfruitful, a combination of **asio** [2] and **cereal** [19] was chosen instead to implement the RPC mechanism manually.

asio (also known as part of the *boost* library as *boost::asio*) is a platform-independent, modern and widely-used networking library written in C++11, providing abstractions for both synchronous and asynchronous network operations. It provides the foundation for the networking code, handling connections and sending of data.

cereal is used to implement the serialization of remote procedure calls on top of *asio*. It is a C++11 library that greatly simplifies serialization of arbitrary C++ data to and from several formats such as JSON and XML. To serialize or deserialize an object of a given class, typically only a single templated function listing all the members of that class needs to be added. This has to be done manually due to the lack of static introspection in C++. *cereal*'s internal binary serialization format was chosen as it is more compact that both JSON and XML and the serialized data is not required to be readable when sending it over the network. After serialization, network data is further compressed using *zlib* [15], a general-purpose compression library.

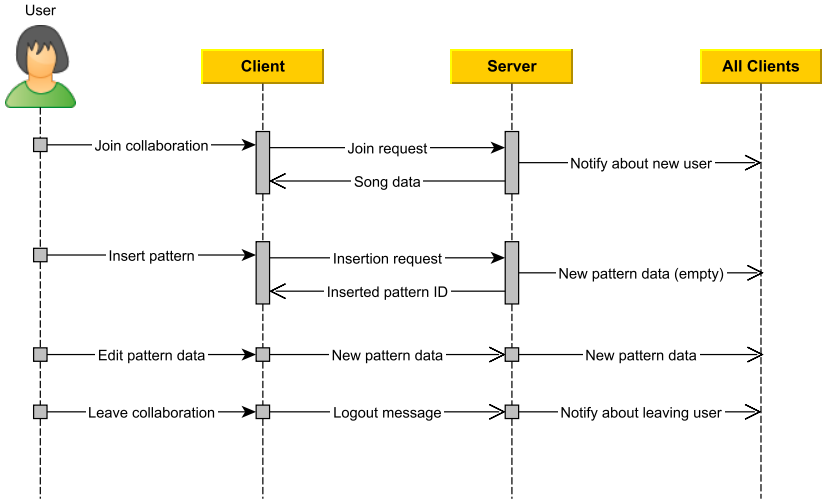

Figure 4.1: Sequence diagram of the different data flow variants

4.1.3 Architecture Details

Client / server: A classic client / server architecture has been chosen for the network code. The user that shares a song automatically acts as the server, and other collaborators or spectators connect to the server as clients. The server acts as a *centralized coordinator process* as defined by Ellis et al. [12]. This makes the implementation of rights management trivial, as the server validates the user rights for each message received from clients before relaying them to other clients. It should be noted in this context that there is little to no other validation of incoming messages and no encryption, as it was not considered to be relevant in the context of this thesis. For real-world usage, the implementation should later be refined to validate messages more thoroughly and guard against out-of-range or otherwise unexpected values, as otherwise an attacker could potentially trigger remote code execution.

Since the architecture requires a peer-to-peer connection, setting up a collaboration session can be moderately difficult in modern network setups: most users have firewalled networks that require explicit whitelisting of TCP and UDP ports to servers inside their network, as otherwise a client cannot make a connection from the outside. There are solutions to this problem such as hole punching [14] and Universal Plug and Play (UPnP), but they were considered to be out of scope for this thesis.

Network byte stream: TCP/IP was chosen as the network protocol for data exchange between clients and the server. While UDP can offer better latency than TCP [7], a reliable, in-order delivery of messages is required for a consistent representation of the music data across all clients. Re-implementing these features on top of UDP to emulate TCP's benefits would thus be pointless.

Synchronization: Most editing actions can be carried out asynchronously. Pattern data is easier to process in this respect than e.g. text data, because editing is normally done in *overwriting* mode rather than *insert* mode and does not shift the document forward or backward at the cursor position. When a pattern cell is edited, its new content can simply be sent to the server asynchronously, and the server then sends the new state to all clients. The client which initiated the change does not have to wait for this process to be finished. Other actions like inserting new patterns need to be carried out synchronously though, as otherwise a conflict would occur if two clients were about to insert a new pattern at the same time. Actions like this are implemented synchronously, where the client waits for the server to provide the ID for a new pattern, instrument or other item to be used. The server sends the content of the newly inserted pattern to all clients, and the client that initiated the request gets an additional notification so that the user interface can be updated, e.g. by jumping to the newly inserted pattern. Figure 4.1 shows the data flow four exemplary actions as they appear in the program. The first two actions, joining a collaboration and inserting a pattern, require synchronization, while the latter two actions are carried out asynchronously.

Data flow: Data sinks implement a simple listener interface that receives incoming data. Both the server and the client are implemented this way, and register themselves as listeners at the `CollabClient`s as seen in Figure 4.2. Both the `RemoteCollabClient` and `LocalCollabClient` extend the `CollabClient` class, the only difference being that the `LocalCollabClient` directly talks to the in-process server without establishing a network connection. When new network data arrives at the `RemoteCollabClient`, it sends it to the registered listener. The client's connection dialog is implemented identically, and changes

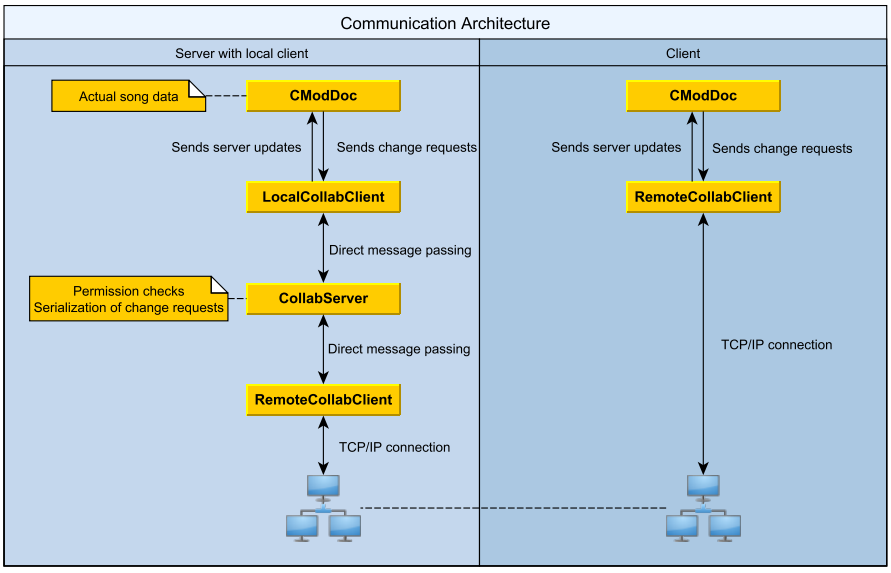

Figure 4.2: Network communication architecture

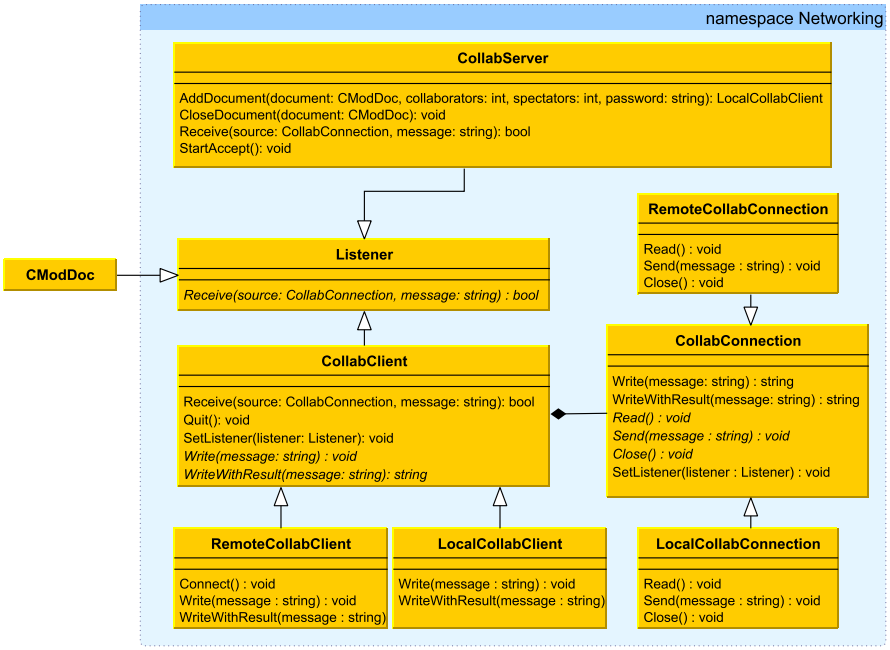

Figure 4.3: The `Networking` namespace containing most of the implementation

the listener to the actual song object once the client joins a shared song. The `CollabClient` instances also receive user updates when the song is changed and pass them on to the server – either directly through a function call on the server side, or through the network connection on the client side.

4.1.4 Code Organization

All code related to network and collaboration was put into a separate namespace `Networking`. The main classes (not including dialogs) of that namespace are documented in Figure 4.3. Outside of this namespace, the main document class `CModDoc` was extended with a listener interface as described above. The architecture required only few other code changes in `CModDoc` and related classes, which were mostly insertions of code for change detection and synchronization. This way, keeping the experimental collaboration code in sync with the main development of *OpenMPT* was relatively simple.

The networking code is mostly contained in the following files:

- `Networking.cpp/.h`: General networking implementation
- `NetworkingDlg.cpp/h`: Sharing, joining, annotation and chat dialogs
- `NetworkListener.h`: Listener interface
- `NetworkTypes.cpp/.h`: Serialization and type definitions for message exchange

In addition, classes that monitor the various song components for changes and initiate the data synchronization, as described in Section 4.2.3, are contained in `GlobalsTransaction.cpp/.h`, `InstrumentTransction.cpp/.h`, `PatternTransaction.cpp/.h`, `SampleTransaction.cpp/.h` and `SequenceTransaction.cpp/.h`.

4.2 Technical Realization

The implementation of the collaboration feature in *OpenMPT* can be roughly categorized into four steps:

1. Implement the client / server architecture
2. Implement the serialization of internal data structures that need to be exchanged
3. Find all user actions that need to be synchronized and implement the data exchange for them
4. Implement any other collaboration-specific features

These steps will be discussed in the following sections. They can serve as a guide for adding collaborative features to any single-user software.

4.2.1 Client / Server Architecture

Both the client and server are implemented directly in *OpenMPT*. In particular, the server is not a separate process, but runs directly in the *OpenMPT* instance of the user that wants to share a song for collaborative editing. As soon as one or more songs are shared, the server starts listening on TCP port 44100 for incoming connections. Network connections are handled using the asio library. When a client connects to the server, their version numbers are compared to avoid incompatibilities between different software versions. If they both run the same version, the server sends a list of shared songs, and the client can then join one of the documents. As a response, the server sends the current state of the song, including all required meta-data such as the list of collaborators, annotations, edit locks, etc. and the collaborative editing can begin.

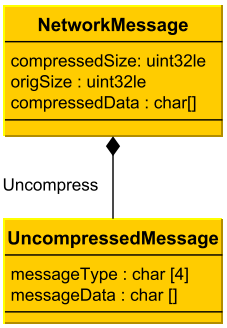

Figure 4.4: Structure of a network message and its uncompressed content.

The protocol used to communicate between client and server is a simple binary protocol with a header (Figure 4.4) containing two 32-bit length fields for the compressed and uncompressed size of the message. The message itself is compressed using *zlib* as a lot of the communication data compresses rather well. The *zlib* state is retained between network messages in order to exploit similarities between messages, leading to a further reduction in network traffic. The decompressed message

(Figure 4.4) consists of a four-byte identifier describing the message type, followed by arbitrary data serialized into a binary *cereal* archive. While the server implicitly identifies clients by their network connection handle, it prepends a unique client ID to each message it sends to the clients so that it is known which change originated from which user.

4.2.2 Serialization of Internal Data Structures

Clients and server need to exchange modifications of the internal data structures, and of course the server also has to send the initial state to all clients when they join.

In *OpenMPT*, all internal song data is contained in a `CSoundFile` object inside the `CModDoc` document object, which contains all the sequences, pattern data, samples, instruments and other miscellaneous data. As a first step of this task, it was attempted to serialize the whole `CSoundFile` object and its contained classes to be able to send the complete initial state to another client. In its most simple form, serializing a class using *cereal* looks like this:

Listing 1: Example of data serialization using the *cereal* library

```
template<class Archive>
void CSoundFile::serialize(Archive &archive)
{
   archive(m_nType, m_nChannels, m_nSamples, m_nInstruments,
      m_nDefaultSpeed, m_nDefaultGlobalVolume, m_nDefaultTempo,
      m_SongFlags, m_nDefaultRowsPerBeat, m_nDefaultRowsPerMeasure,
      ChnSettings, Patterns, Order, Samples, m_MidiCfg, m_MixPlugins,
      m_songName, m_songArtist, m_songMessage, m_madeWithTracker,
      ...
   );
}
```

This single function can serve both for serialization and deserialization of the data, depending on the template parameter `Archive`. Primitive data types such as integers, but also many useful containers from the C++ standard library (such as `std::string` and `std::vector`) are handled automatically by *cereal*, but since our `CSoundFile` object is also composed of many custom data types such as patterns and instruments, custom serialization functions had to be written for all of these classes as well. Once this was done, the complete compound object could be serialized in just one function call.

However, serialization of some of the contained objects was not directly possible at first, as *cereal* cannot serialize raw pointers. This is due to the fact that a raw pointer could point to any number of objects and de-duplication of pointed-to objects can be difficult when trying to serialize them. As a result, *cereal* can only serialize smart pointers such as `std::shared_ptr` [20]. Like many applications with legacy code, some objects (patterns, sequences and sample data) were managed using raw pointers to dynamic arrays. Turning pattern data and sequences into `std::vector` objects not only made serialization of these objects considerably easier, but it also greatly simplified the implementation of their public interface, as e.g. insertion and deletion of pattern or sequence data

could now be handled directly using the `std::vector` implementation rather than through custom functions. All in all, the modifications made this part of the *OpenMPT* source code safer and more future-proof, so these modifications were directly merged back into the *OpenMPT* main repository. Sample data was handled separately through `cereal::binary_data` instead, as it would have been considerably more work to use `std::vector` there due to some "pointer tricks" being used in the low-level audio mixing routines. `cereal::binary_data` allows to directly send any binary object by specifying a pointer and size. This also avoids creating duplicate copies of sample data when sending only parts of a modified sample over the network, as we can directly point into the sample data rather than having to replace it completely.

Most serialization and deserialization functions can be found in the file `NetworkTypes.h`.

4.2.3 Synchronization of User Actions

Now that all the internal data structures can be synchronized, they have to be monitored for modifications, so that synchronization can be triggered when required. In the most simple case, only modifications initiated by the user need to be identified, but in more complex applications, there may also be other sources that can manipulate the data structures, such as plugins or scripts. In most applications, there are several ways of identifying user actions that modify the document:

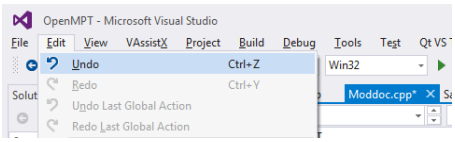

Figure 4.5: Undo buffer, and modification status of an open document (`Moddoc.cpp`) in Visual Studio.

- Common programs mark a document as modified, sometimes denoted by adding an asterisk to the document's name in the title bar as seen in Figure 4.5. Typically, there is either an explicit call to a function (called `SetModified()` in *OpenMPT*) in the code that handles the user action, or the edited object automatically calls this function whenever one of its setter methods is called.

- Typically, programs also provide an undo buffer that records the state of the edited object (e.g. some note data) before the modification has been carried out. In *OpenMPT*, this is done for samples, patterns and instruments by calling `PrepareUndo()` on the sample, pattern or instrument undo buffer.

- Programs that expose a sophisticated scripting engine to the user may already allow to register "watcher" functions that are automatically called when an object is modified. This third method is not used in *OpenMPT*, as it currently does not have a scripting interface.

- If none of the previous options are available, every single editing action needs to be reviewed manually. Except for very minimalist or experimental software, this should never be the case, though. Manual testing of all

editing actions can still be beneficial to detect inconsistencies such as forgotten function calls to `SetModified()`.

To identify the places where the code had to be modified, a mixture of the first two approaches was chosen by finding all references to `SetModified()` and `PrepareUndo()` functions. As a side effect, this activity also unearthed several cases where the two functions were not called consistently, e.g. by only marking the document as modified or only preparing the undo buffer instead of doing both. These problems were fixed directly in the main *OpenMPT* repository.

After all points of user actions had been identified, a way of detecting and sending the changes had to be developed. To avoid having to write specific code for every user interaction, a few simple classes were implemented for all the `CSoundFile` child classes that follow the well-known **RAII** C++ idiom (Resource acquisition is initialization): in their constructor, they take a snapshot of the to-be-modified object (such as a selection of pattern data), and in the destructor the object's new state is compared against the copy. If there were any changes, the difference is sent to the server automatically. This means that only a single line of code has to be added to each user action, as seen in the following example:

Listing 2: Example of detecting and transmitting user changes

```
1  // User has changed the default volume of a sample (simplified example)
2  void CCtrlSamples::OnVolumeChanged()
3  {
4     int volume = GetDlgItemInt(IDC_EDIT7);
5     ModSample &sample = m_sndFile.GetSample(m_nSample);
6     if (volume != sample.nVolume)
7     {
8        SamplePropertyTransaction transaction(m_sndFile, m_nSample);
9        PrepareUndo("Set Default Volume");
10       sample.nVolume = volume;
11       SetModified(SampleHint().Info(), false, false);
12    }
13 }
```

In line 8, a `SamplePropertyTransaction` object is created on the stack which records the current state of the sample slot. After the undo buffer has been updated, a single property of the sample is then modified. At the end of the scope (line 12), the destructor of the transaction object is automatically called and sends the new state of the object to the server if anything changed.

In addition to these checks if an object changed at all, there are also more fine-grained checks employed on pattern data. As many pattern operations span a rectangular selection that can potentially cover areas edited by other users, only pattern cells that really were modified by the current user should be transmitted. This avoids overwriting the changes made by other users in the same area at the same time. The server then only updates its own pattern data with the changed cells, but sends the full rectangle back to all users as an authoritative data source.

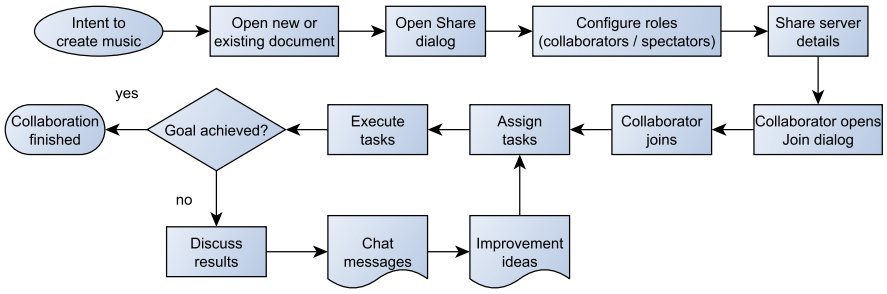

Figure 4.6: Workflow of the collaborative *OpenMPT* implementation.

Change detection is not the only requirement for keeping the song data consistent. As the network code runs in a separate thread in order to not stall the user interface, access to internal data structures needs to be synchronized in some way. The easiest, most generic but not necessarily most efficient way to do so is by means of locks. In *OpenMPT*, there are already locks around most editing actions, in particular those that access resources that can be added and removed dynamically (such as patterns, samples and instruments), as the user interface thread already needs to be synchronized with the audio rendering thread. However, even when such locks are not yet present, they can be trivially integrated into pattern described in Listing 2: the lock guard can be a member of the `SamplePropertyTransaction` class and thus guard the whole transaction. Naturally, a lock guard also needs to be inserted into the code that deals with the data received from the network thread.

Instead of locks, it is also possible to use atomic values for simple global data: access to integer or floating-point properties does not need a lock if it does not result in the modification of dependent values. As long as the read and write is atomic, the data is already safe and consistent. Applying lock-free updates to larger data structures, however, is error-prone and not trivial to implement, so locks are the easier and safer solution in that case.

Depending on the software architecture, there is yet another possible implementation for the data transfer from the network thread: instead of applying the updates directly in the network thread, a message-passing mechanism that works across threads (such as window messages in *Windows* using `SendMessage` or `PostMessage`) can be used to transfer the messages in the user interface thread. This way, synchronization only needs to happen between the user interface thread and audio thread, as before the networking implementation.

4.2.4 Collaboration Workflow

The collaboration workflow resulting from this implementation, as described in Figure 4.6, is supported by two dialogs for sharing a song and joining a collaboration.

In the sharing dialog (Figure 4.7), the user who intends to share a song chooses how many collaborators and spectators can join them. The collabo-

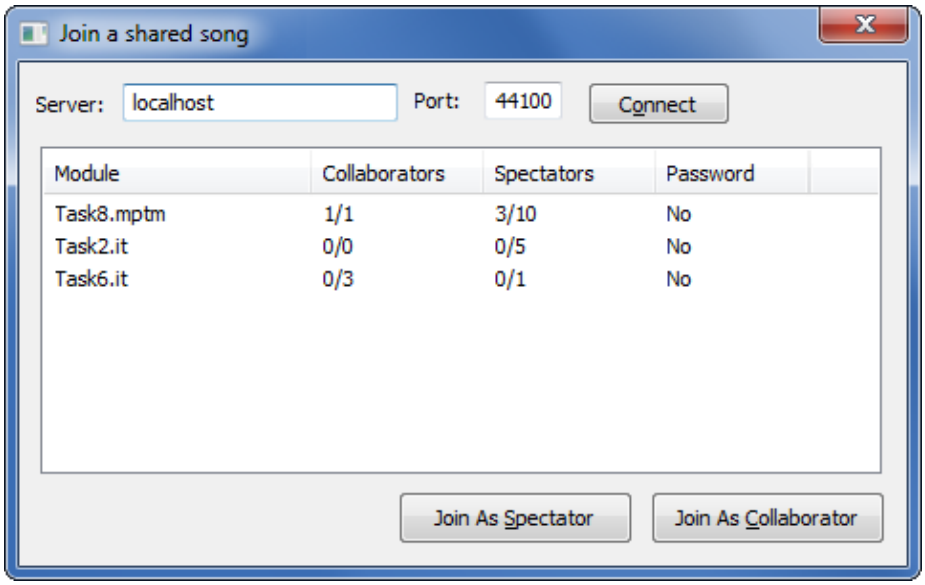

Figure 4.8: Dialog for joining an existing collaboration.

ration can optionally be secured by a password. As soon as the user hits the "Share" button, the collaboration server is started and accepts requests from clients. If required, all properties in this dialog can be changed later while the collaboration is running, e.g. to invite more collaborators. Collaboration automatically ends when the song is closed.

Collaborators and spectators can join a shared song by opening the Connection dialog as seen in Figure 4.8. In this dialog, they can specify a server (by means of a host name) they want to connect to. After the connection has been established, they see a list of shared songs, including information about how many people can still join the collaboration and whether a password is required. After marking a song in the list, the user can join the document either as a collaborator or as a spectator by clicking the appropriate button. Leaving a collaboration is as simple as closing the document.

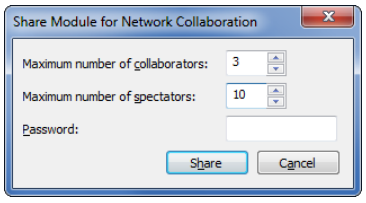

Figure 4.7: Dialog for sharing a song with other users.

Once the shared song is loaded, users can start editing as usual. There are no differences they have to be aware of, apart from the additional features that were implemented and are presented in the next section.

4.2.5 Additional Collaboration Features

In the following section, the implementation of additional collaboration features introduced in the third chapter that surpass the simple synchronization of data

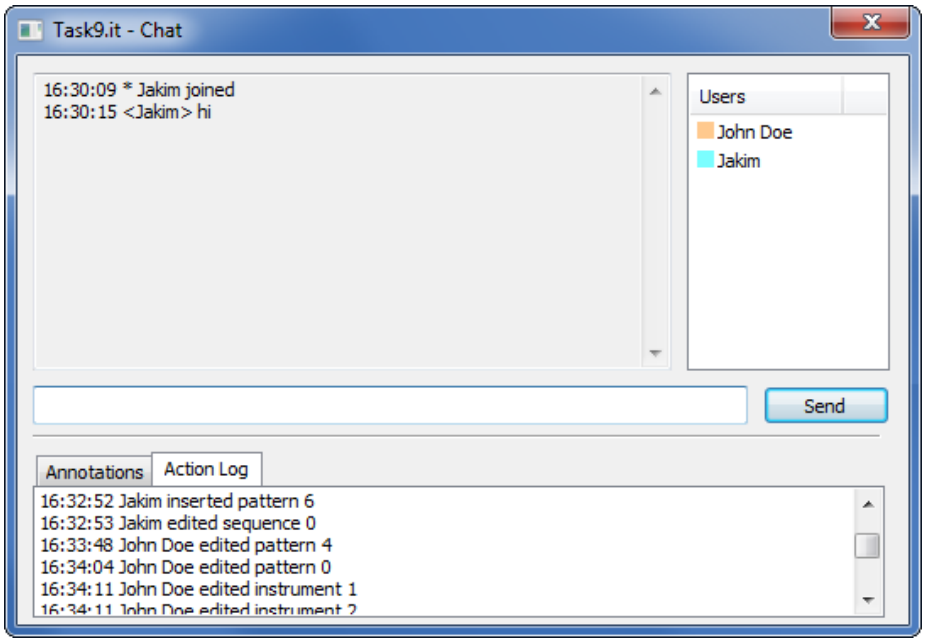

Figure 4.9: Chat window with user list and action log.

is discussed and reasoned about.

1. Editing: The various data structures that make up a song are synchronized in different ways described in Section 4.2.3.

2. Chat: The chat window, as seen in Figure 4.9, is the communication hub for the collaborators. Besides offering a simple plain-text chat, a list of all participants including their edit cursor color (see feature 4 below) is displayed, which is a central part of the communication workflow. In addition, the window contains the annotation list (see feature 7) and an **action log**.

The action log offers a very condensed view (showing what has been edited, but not how) of the collaborators' actions. Whenever a collaborator starts editing a new entity (e.g. a different pattern), it gets added to the log (e.g. "John Doe edited pattern 1"). The combination of features in the Chat window allows to keep track of the song changes on a high level and directly discuss them if needed. Another reason for using this dialog for several features was to avoid having too many additional windows open and reduce screen clutter (in accordance with the second law of the Ten Laws Of Simplicity: Organize [29]).

3. Rights management is used to allow or deny certain actions for some users. Two rights levels were implemented: collaborator and spectator. Due to the client-server architecture, all verification can be carried out by the server before potentially relaying incoming messages to other clients. Every message received by the server is associated with a specific client object, and all messages that would modify the song state are ignored if the client object belongs to a spectator. Other messages such as chat messages are still forwarded to all clients.

In a similar fashion, it would be possible to implement more fine-grained rights management in order to only allow certain editing actions to be carried out by specific collaborators.

Additionally, shared songs can be protected by a password so that only authorized collaborators and spectators can join a song. The password is checked when a client wants to join a shared song. As mentioned in the description of the architecture, information security was not considered to be within the scope of this thesis, so there is no encryption and the password is transmitted in plain-text.

4. Shared edit cursors: When the position of the edit cursor is changed through the function `CViewPattern::SetCurSel`, the collaborator's client notifies the server of the new position. The server then broadcasts the position to all other clients. Figure 4.10 demonstrates the visualization of shared edit cursors: in the pattern grid, the user's own cursor is displayed as usual as a black rectangle, while other users' cursors are shown in the respective user color (row 3 and 7 in this example). Hovering a pattern cell that is edited by another user with the mouse shows a tool-tip containing the user's name.

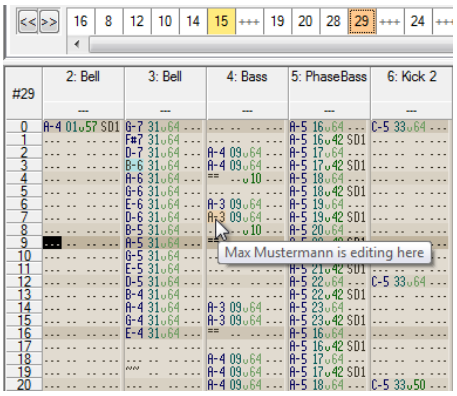

Figure 4.10: Collaborators' shared edit cursors

The order list above the pattern grid contains the play order (or sequence) of all patterns that form the song. Similarly to the pattern data itself, the order positions that other users work on are colored in the users' respective colors, giving a rough idea what everyone is currently working on. Hovering these order positions also reveals the user's name.

5. Automatically following edit cursors: This feature was only implemented in spectator mode, but could also be applied to collaborator mode. By clicking on a collaborator's user name in the user list, a spectator starts following the collaborator's edit cursor automatically. That way, the spectator sees almost exactly what the collaborator is seeing (modulo different display settings), similar to a video stream. Playback actions such as play and pause are also sent to spectators to complete the experience.

6. Edit Locks can be used to restrict the editing of a specific pattern to a single user. This way, it is guaranteed that no pattern data is overwritten. The rights management for this feature is implemented both on the client and the server:

1. The client that wishes to establish or release an edit lock sends this intention to the server.
2. The server checks if there is already a lock held by a different client. If

this is not the case, it notifies all clients that the pattern is now locked.

3. Clients check if the pattern the user wants to edit is already locked, and display an unobtrusive warning if this is the case (Figure 4.11).

4. Since an edit lock request from another client may arrive at the client after sending its own edit action for the same pattern, the server rejects any edit actions from other clients after a lock has been placed and sends back the original pattern content so that they can undo their edit action.

In future versions, this feature could automatically follow the currently edited pattern so that a user can always prevent other users from editing the same pattern, without the need to manually update the locks all the time. Locking a range of pattern channels could also help if a user is only interested in keeping collaborators from editing certain instruments. This way, they could still edit different instruments in the same pattern, if required. Again, similar features could be added to samples and instruments.

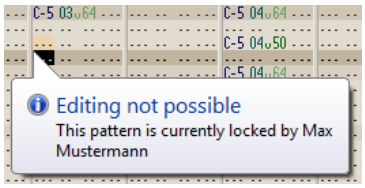

Figure 4.11: Pattern locked by a collaborator.

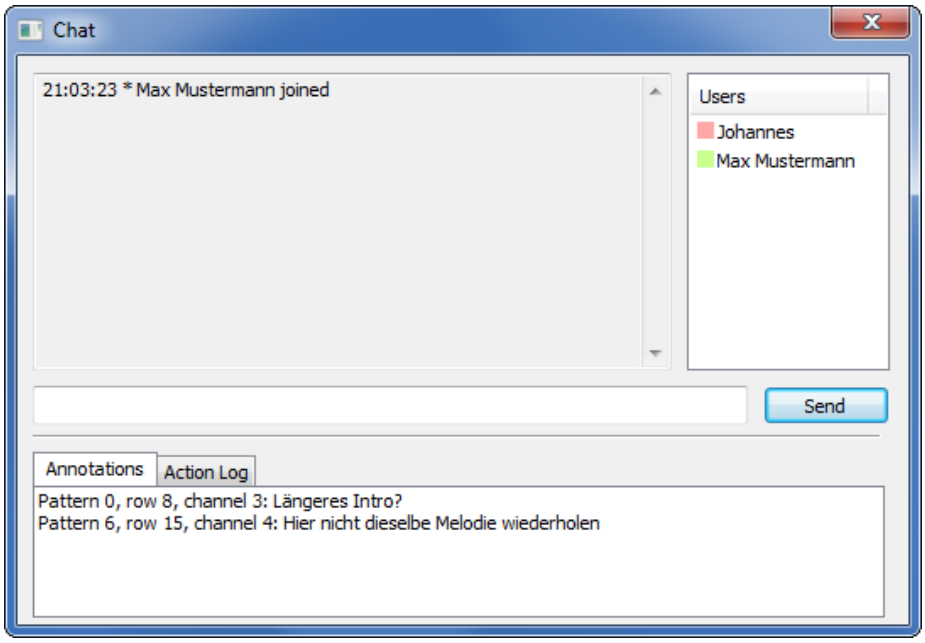

Figure 4.12: Annotations are shown in the chat dialog to reduce clutter.

7. Annotations can be added to any position in the pattern data and are permanent, as opposed to chat messages that only exist during the editing session and are lost when the song is closed. As annotations are supposed to facilitate discussion, they are not user-specific. Every pattern position can have at most one annotation, but more than one user can edit that annotation. In order to be not too visually intrusive, pattern cells with annotations are displayed with a dotted border (Figure 4.13). When hovering such a cell with the mouse, the annotation text is shown, and the annotation editor can be opened through a context menu entry. To be able to quickly locate all annotations, a list of all pattern locations with annotations is provided in the chat window (see Figure 4.12) for quick access. Clicking on an annotation directly jumps to the pattern containing the annotation so that its context can be examined.

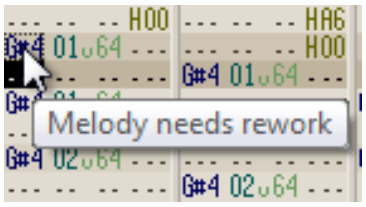

Figure 4.13: Annotations are displayed in the pattern in an unobtrusive way.

5 Analysis

The implementation of the collaborative music software was analyzed through a user study. Since the implementation is based on an existing music software, it made little sense to rate the usability of the software as a whole, in particular since the software is quite complex already without the collaboration extensions. There is no baseline and the implementation cannot directly be compared to other software, so Nestler et al.'s approach [35] was followed to design the experiment. The approach itself is based on the Technology Acceptance Model [11] and other usability questionnaires. In the end, we can judge the usability and effectiveness of the collaborative mode and the new collaboration features that were added, and find possible issues in the implementation.

5.1 Subjects

The experiment was conducted with 17 *OpenMPT* users of varying expertise who were already familiar with its interface. To be able to judge the collaborative aspect of the software under realistic circumstances, users located in several countries participated in the study: six users from Germany, four users from the USA, two users from Norway and one each from Ireland, Israel, the Netherlands, Poland and South Africa. 16 subjects are male and one is non-binary. Five users are students, four are software developers, two work in IT and the others are a scientist, an electrical engineer, an information security consultant, a customer service associate, a graphic designer, and a mathematician respectively. The subjects are 21 to 41 years old, and all of them have experience with music software. The majority of the subjects (10 out of 17) is already familiar with collaborative software.

The amount of participants exceeds the minimal number of six subjects required for a statistical significance of usability tests [21].

5.2 Experimental Setup and Data Collection

All subjects were provided with a copy of the *OpenMPT* version with collaboration support. The experiment was conducted locally with two expert users, while the actions of the remaining users were monitored through screen-sharing software to verify if the tasks were carried out correctly. Since all subjects were already familiar with the software and the user interface additions were described in the tasks for the experiment, no training had to be performed before the evaluation.

The experiment consists of three parts:

1. A set of nine collaborative tasks to perform together with me.
2. A questionnaire with ten questions to evaluate the results from the tasks.
3. Three open-ended questions that also served as a basis for discussion with three expert users.

1. Tasks: The users were asked to fulfill nine tasks addressing all of the new collaborative features. The last two tasks were combined tasks, testing

a sequence of actions. Users were asked to make use of a variety of different editing actions for the evaluation. Here is a short summary of the tasks:

1. Connecting to a server, listing shared documents
2. Evaluating the chat functionality
3. Concurrent pattern editing
4. Following collaborator changes
5. Sharing a new song
6. Concurrent editing with pattern locks
7. Annotations
8. Combined task: Connecting, concurrent pattern editing, instrument editing, sample editing
9. Combined task: Connecting, instrument creation, instrument editing, pattern locking

The detailed task descriptions can be found in Appendix A.

2. Questionnaire: After carrying out the assigned tasks, all users answered a set of ten questions derived from Appendix 1 in [35] and rated on the five-point Likert scale (1: I strongly disagree, 5: I strongly agree). The questions are sorted into five categories defined by Nestler et al.: Utility (U), Intuitiveness (J), Memorability (M), Learnability (L) and Personal Effect (P). None of the questions from the memorability category were relevant to the evaluation, so no questions were chosen from this category. The chosen questions are:

1. The software is reliable. (U)
2. The software is cumbersome to use. (U)
3. The software enhances my effectiveness. (U)
4. The software meets my expectations. (J)
5. I feel comfortable using this software. (P)
6. The software is frustrating to use. (P)
7. I would like to use the software in the future. (P)
8. The software is mentally demanding. (P)
9. The software facilitates performing my tasks. (U)
10. The visual feedback is appropriate. (L)

Note that "the software" refers to the collaborative features in particular, and not to *OpenMPT* in general. A usability factor can be derived from these questions.

3. Open-ended questions: The remainder of the experiment consisted of three open-ended questions:

1. What do you think about the software's reliability?
2. Which parts of the software could be improved specifically for collaborative use? Which collaboration-specific features are missing that could help enhancing the workflow?
3. To what extent does the software enhance your effectiveness and productivity? Does it meet your expectations?

Three expert users were invited to participate in a discussion where they could elaborate more on these three questions. This number of expert users was expected to be enough to pinpoint the most obvious problems with the software, as most usability problems can be detected within three to five subjects [35]. Other users were not required to answer these questions, but were encouraged to do so.

5.3 Data Analysis

The collected data was analyzed in three steps corresponding to the experiment setup: tasks, questionnaire and open-ended questions.

1. Tasks: For every task, the time required to fulfill the task and the correctness of the solution were recorded. The user's score on a task was rated on a scale from 0 to 1 depending on the correctness and accuracy of the answer. Tasks with several steps were averaged over the correctness of the individual steps.

With this data, it would be possible to compute a speed-accuracy trade-off score, but it was not considered to be a sensible metric for this evaluation: regardless of the correctness of the result, it was observed that the time taken to solve many of the editing tasks largely varied between users, as some just fulfilled the bare minimum of editing for solving a task, while other users tested the editing features more thoroughly, thus taking more time to solve the tasks. In this case, the resulting calculated usability would be lower, even if the actual outcome of the task did not reflect this. Hence, the time dimension was discarded from the evaluation and only the correctness of the tasks was considered as a metric.

2. Questionnaire: There were four questions in the Utility (U) category, four questions rating the Personal Effect (P), one question regarding the Intuitiveness (J) and one rating the Learnability (L). Different weights can be assigned to the categories to shift focus to particular usability concerns, and in fact the questions in each category can be sorted into weighted sub-categories as well. In Nestler's example [35], questions regarding stress are rated higher when evaluating user interfaces for emergency situations. In the context of this evaluation, a balanced weighting with no specific focus was considered to be reasonable.

From these scores, the usability score U over the categories C with weights $w(s)$ and scores $v(s)$ can be calculated:

$$U = \sum_{c \in C} \left(\sum_{s \in S(c)} w(s) * v(s) \right) = \sum_{c \in C} \left(U_c * \frac{w_c}{100} \right)$$

3. Open-ended questions: Nestler [35] also proposes to rate open-ended questions from expert discussions on a three-point scale: answers can be categorized as positive (1.0), neutral (0.5) and negative (0.0). The results could then be handled exactly like those of the questionnaire and the result could be quantified. However, it was decided that these scores would not be very meaningful, as e.g. everyone provided improvement suggestions, so technically each comment on the second question would need to be rated neutral. Instead, the focus was shifted to the qualitative aspects of the answers and will be reflected in more detail in the discussion in Section 6.1.

5.4 Results

1. Tasks: The averaged scores and durations per task can be found in Table 1. All in all, the average time required to solve all tasks is 30 minutes.

It can be observed that most tasks apart from Task 4 (following collaborator changes) were solved with high accuracy. The most prominent issue was the identification of the exact edited pattern locations, in particular when existing notes were edited rather than inserting new notes. Hence, this feature should be focused on when analyzing the overall usability result. Nevertheless, a definite increase in efficiency can be noted compared to the old method of exchanging collaborations via mail: locating changes becomes much easier even if only the approximate position is known, compared to having to re-listen the whole song to find changes after receiving it from a collaborator.

Task 6 (concurrent editing with pattern locks) achieved the second-lowest average score: while all participants easily found out how to lock patterns, it was not always clear how to find out which other patterns were locked. As a result, locked patterns should be visualized more clearly.

Task 1 (finding shared songs) had a similarly low score, but only because some users hastily listed the shared songs and forgot to mention the number of allowed collaborators and spectators for each document. It is unlikely that there is a usability problem.

Apart from the two mentioned issues, no major repeating mistakes were identified in the other assignments. Simple tasks such as finding shared songs or annotations were executed quickly on average, so most likely their implementation is acceptable. The users spent most of the time writing down the solutions when solving these tasks, which is included in the execution time.

Table 1: Task scores and execution times

Task	1	2	3	4	5	6	7	8	9
Score	0.853	0.971	0.985	0.647	0.941	0.853	1	0.926	0.897
Time	1.647	1.176	2.412	5.176	2.235	3.647	3.588	5.647	4.375

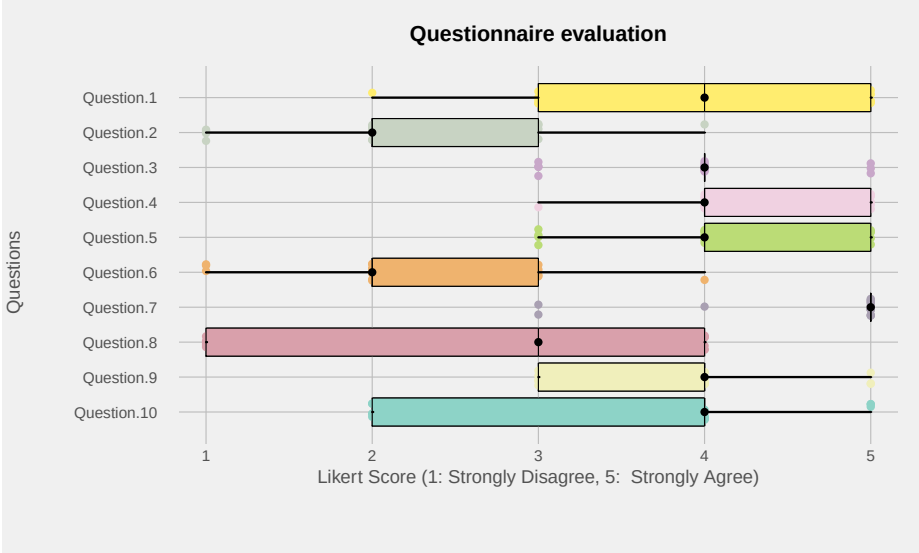

Figure 5.1: Boxplot showing the data distribution of the questionnaire.

2. Questionnaire:

Table 2 shows the average scores for all questions. Note that questions 2, 6 and 8 were formulated negatively, so their average scores were inverted for the computation of the category scores. Figure 5.1 shows a boxplot based on the original questionnaire data.

Table 2: Questionnaire scores

Question	1	2	3	4	5	6	7	8	9	10
Score	3.882	3.882	3.941	4.294	4.059	3.824	4.706	3.353	3.882	3.412

From these scores, a usability factor U can be derived as described in Section 5.3.

Table 3: Calculation of the usability score U of the collaborative editing mode.

Category	U	J	P	L
$w(s)$	0.25	0.25	0.25	0.25
$v(s)$	0.724	0.824	0.746	0.603
$w(s) * v(s)$	0.181	0.206	0.187	0.151
$\sum$	0.724			

Table 3 shows the obtained values from the questionnaire and the resulting usability score $U = 0.724 \in [0, 1]$.

This score is far from optimal, and in particular the scores for question 8 (The software is mentally demanding) and 10 (The visual feedback is appropriate)

indicate that collaborative features need even more attention to visual support in the user interface in order to be easier to use.

3. Open-ended questions: Although only required to be filled in by the three chosen expert users, all users answered at least one of the open-ended questions. The evaluation of their answers showed that the expert users' usability concerns were mostly echoed by the other users. As mentioned, the average scores on the questions (0.75, 0.5 and 1.0 respectively) were not very meaningful or did not provide any new insight compared to the previous sections, so a quantitative result cannot be deduced from the open-ended questions. The replies can be summarized as follows:

1. The feedback regarding reliability was mixed depending on the stability of the software during the experiment. Half of the experiments were carried out without any problems, but four sessions were interrupted by crashes, while the rest was only affected by minor issues. One of the users that experienced crashes had a very unstable internet connection, which was also mentioned in the context of this question. The software reliability has to be improved in particular in the context of unstable network connections.

2. A lot of useful suggestions were made in the answers to the second question. They will be discussed further in Section 6.1.

3. In general, it can be said that the feedback was positive and people enjoyed the experience of making music collaboratively. The expectations of some participants were even exceeded.

6 Discussion and Conclusion

In the remainder of the thesis, we will discuss the results from the user experiment and summarize the results.

6.1 Discussion

While the collaborative implementation was targeted to make all single-user editing actions available to collaborative editing, this is obviously not enough to provide the highest possible satisfaction for collaborators, as the mediocre usability score of $U = 0.724$ shows. Several collaborative tools were added to the software, such as the chat functionality and shared editor cursors, but more collaborative features should be added and the existing features should be improved in the future to make the software easier to use. The discussion with expert users resulted in the following suggestions, backed up by the feedback from the other users. The users' replies are summarized and paraphrased.

The most common criticism uttered in the questionnaire addressed the **visual feedback**:

- Other users' editing cursors should stand out even more, for example by marking the entire edited row in the user's color.
- Most commonly suggested, recently edited data should be marked in the user's color, slowly fading away over time. This way, it is not necessary to keep track of the action log to get a rough idea what the other users are doing, in particular since the action log is not very detailed.
- Due to time constraints, locked patterns were not visualized properly. The only way to know if a pattern is locked is by trying to edit it, and by watching the action log. A lock icon can be added to the locked patterns in the order list to visually indicate their status.
- The raw action log needs to be represented in a more visually pleasing way, for example as a data grid with different groups for patterns, samples and instruments.
- Joining a collaboration can take a while, since a lot of data needs to be transmitted for some songs. Currently there is no indication how much data is left to transfer, so the user cannot know when the loading process is done. A progress bar needs to be added to indicate the downloading status.
- It is easy to miss incoming chat messages, because the only visual indication is the new message itself. The user must, at least optionally, be notified so that they do not forget about those messages.
- It should be possible to follow the edit actions of another musician not only as a spectator, but also as a collaborator, making it easier to follow their sketches and ideas.
- Not everyone considered the newly added icons to be meaningful, so better icons need to be found.

Other feedback addressed technical issues:

- For some users, editing instrument envelopes was very cumbersome due to the round-trip-time from the server. When continuously moving an envelope point with the mouse, the received envelope point position from the server lags behind the mouse cursor, in particular if the two collaborators are not geographically close. This shows that for some parts of the application, the approach of a *centralized coordinator process* does not work well and an alternative like Ellis' Operational Transformation approach [12] or Sun's [43] and Cormack's [9] improvements must be explored.
- One participant of the experiments had a very unstable internet connection, causing their client to be disconnected from the server several times. Clients could reconnect automatically and transparently in this case.

But there was also a lot of positive feedback:

- Most importantly, the immediacy of real-time collaborations was described as invigorating and motivating, leading to new sources of inspirations, increased efficiency and better flow for cooperations.
- Apart from producing full compositions, further usage scenarios were suggested such as drafting, or using it as a training tool for teaching how to use the software or how to write music.
- The annotation feature was deemed to be very useful even in single-user contexts and should be expanded to be able to annotate any part of a song, not just the note data.

In addition, the following observations were made during the experiment:

- People instantly figured out how to lock patterns, indicating that the feature was placed intuitively.
- The buttons for sharing and joining shared songs were typically found rather quickly as well.
- While some participants instantly realized how to annotate their patterns, others did not find the functionality instantly and searched in unrelated parts of the program. However, this confusion was possibly caused by the assumption that annotations are unique per pattern, while they can actually be used to annotate specific pattern cells.
- Most information for solving the tasks was found in the action log, but it was not always used. It needs to be made more prominent and usable, as mentioned above.
- Managing screen estate was sometimes cumbersome due to the extra windows, in particular on smaller screens. Moving some of the information to the main window could help with further reducing screen clutter. It was suggested to make the chat window dockable, and the action log and annotation list should be resizable.

The workload for each step of the collaboration task model is greatly reduced in particular due to the avoidance of context switches to other supporting applications:

- **Task execution:** Musicians no longer need to take turns at editing the file but can realize their ideas immediately. The danger of breaking their flow and forgetting ideas until they are allowed to edit again is avoided.
- **Data exchange and synchronization:** Song files no longer have to be sent back and forth between collaborators. Changes can be observed live, rather than having to scan the entire song repeatedly. With the improvements suggested by the expert users, following collaborator changes can be simplified even further.
- **Comments and discussion:** It is no longer necessary to switch to an external chat application, which can break the flow and creativity. Changes and ideas can be directly annotated in the music software now. It is not required to note down the exact location of what needs to be discussed, which can be error-prone and cumbersome.
- **Rights management:** Roles can be directly assigned at the start of the collaboration, not requiring any further discussion. Tasks can change dynamically during the composition, which is supported by the integrated chat.

All in all, expert users agreed that the possibility to collaborate with musicians in other geographic locations was very useful to them and the functionality even surpassed their expectations. For example, it was mentioned that the software will help them working with a more skilled musician to help them with their musical drafts where required. Not having to wait for the other musicians, as with the "classic" way of collaborating, is a great enhancement to some users' workflow. The spectator mode in particular was very novel and interesting to them.

6.2 Conclusion and Future Work

In this thesis, a collaborative editing mode was added to *OpenMPT*. It already supports the basic collaboration tasks quite well, so that it is possible to collaborate with another tracker musician in real-time. Real users of the software accepted the implementation, which even surpassed the expectations of some people.

In addition to the improvement suggestions from the usability evaluations, there is a number of other technical improvements that should be made and ideas that need to be researched:

- **Plugins** are local resources that cannot be shared easily. A plugin consists of executable code (a shared library) and possibly arbitrary other files, and may require proper installation in order to function. Commercial plugins often also need to be licensed on the computer they run on. This means that the music software cannot simply share missing plugins between collaborators, but when inserting new plugins into a song, the

choice of plugins could be limited to those that are available to all other collaborators as well. This ensures a more consistent playback between all collaborators.

- To allow more flexible sharing, the server component should, at least optionally, not be required to run in the same process or even physical computer as the *OpenMPT* instance the song is being edited in. This would allow for songs to be automatically shared over an extended period of time and not require the user who shares the song to be available at all times. A central Network Information Service (a "yellow pages server") could also serve as a general hub for musicians to join any collaborations projects by fellow artists they are interested in.

- Collaborative features that were only implemented partially for the purpose of demonstration need to be extended: edit cursors and locks for sample and instrument slots are currently lacking, and the granularity of pattern locks needs to be revised.

- Another long-standing feature request for *OpenMPT* is a scripting API; Both the collaborative mode and the scripting functionality add similar complexity to the code, in particular regarding synchronization of concurrent access. As described in Section 4.2.3, the collaborative features can make use of the scripting infrastructure. To reduce the overall complexity of the code, an implementation of the collaborative mode entirely using the scripting API should be researched.

- It should be investigated whether it is sensible to replace the client-server approach with the research by Sun [43] and Cormack [9].

- It should be researched if and how smartphones or tablets can support the process of collaborative music composition, e.g. by showing the content of the Chat window on a second screen, or by displaying a visual outline of the song including locks, annotations and changes done by collaborators.

In general, we have seen that it is possible to extend the workflow of trackers to more than one user, and the fast edit-audition feedback cycle leading to a high degree of liveness is preserved or even amplified when several collaborators work on tracked music at the same time. Musicians no longer have to interrupt their workflow by sending the current status back and forth and discussing it in external chat programs. Instead, they can rapidly audition their collaborators' changes and ideas and get inspired by them in completely new ways. The creative process is enhanced by the immediacy of real-time collaboration and supported by long-term features such as annotations. Collaborative music software does not only enable musicians to work on a joint composition, but it can also be used in an educational context.

On the technical side, the results from this implementation can also be generalized to other single-user software, as seen in Section 4.2.3 in particular. While the choice of an optimal synchronization scheme (centralized coordinator process, operational transformation, etc.) depends on the data that needs to be synchronized, the change detection will often look similar and can be easily adapted for the necessary data synchronization. In C++, this can be achieved

by adding a single line of code at each modification site thanks to the RAII principle.

With the insight about change detection and synchronization obtained from these steps, it should be moderately easy to extend another tracker with collaboration features, although the steps can also be generalized to sequencers and completely different kinds of software.

References

[1] Ableton AG. Ableton Live. https://www.ableton.com/en/live/, retrieved on 10 January 2018.

[2] asio authors. asio C++ library. https://think-async.com/, retrieved on 3 January 2018.

[3] Peter Barth and Chris O'Neill. MilkyTracker. http://milkytracker.titandemo.org/, retrieved on 10 January 2018.

[4] R Beuscart, C Grave, M Wartki, S Serbouti, and MC Beuscart-Zephir. Computer supported cooperative work for medicine imaging. In *Engineering in Medicine and Biology Society, 1992 14th Annual International Conference of the IEEE*, volume 3, pages 1213–1214. IEEE, 1992.

[5] Blend.io. Blend - Make Music Together. https://blend.io/, retrieved on 20 December 2017.

[6] Paul Booth. *An Introduction to Human-Computer Interaction (Psychology Revivals)*. Psychology Press, 2014.

[7] Kuan-Ta Chen, Chun-Ying Huang, Polly Huang, and Chin-Laung Lei. An empirical evaluation of TCP performance in online games. In *Proceedings of the 2006 ACM SIGCHI international conference on Advances in computer entertainment technology*, page 5. ACM, 2006.

[8] Sook Kuan Chin and Alvin W Yeo. Computer Supported Cooperative Work (CSCW) in orthography system development. In *Computer Applications and Industrial Electronics (ICCAIE), 2010 International Conference on*, pages 579–583. IEEE, 2010.

[9] Gordon V Cormack. A calculus for concurrent update. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, page 269. ACM, 1995.

[10] Tracktion Corporation. Tracktion Music Production Technology. https://www.tracktion.com/, retrieved on 10 January 2018.

[11] Fred D Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS quarterly*, pages 319–340, 1989.

[12] Clarence A Ellis and Simon J Gibbs. Concurrency control in groupware systems. In *Acm Sigmod Record*, volume 18, pages 399–407. ACM, 1989.

[13] Clarence A Ellis, Simon J Gibbs, and Gail Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[14] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *USENIX Annual Technical Conference, General Track*, pages 179–192, 2005.

[15] Jean-loup Gailly and Mark Adler. zlib Home Site. https://zlib.net/, retrieved on 6 January 2018.

[16] Steinberg Media Technologies GmbH. Steinberg Cubase. https://www.steinberg.net/en/products/cubase/start.html, retrieved on 10 January 2018.

[17] Steinberg Media Technologies GmbH. VST Connect Pro. https://www.steinberg.net/us/products/vst/vst_connect/vst_connect_pro.html, retrieved on 20 December 2017.

[18] Google Inc. gRPC, a high performance, open-source universal RPC framework. https://grpc.io/, retrieved on 3 January 2018.

[19] W. Shane Grant and Randolph Voorhies. cereal - a C++11 library for serialization. https://uscilab.github.io/cereal/, retrieved on 3 January 2018.

[20] W. Shane Grant and Randolph Voorhies. Pointers and References. cereal - a C++11 library for serialization. http://uscilab.github.io/cereal/pointers.html, retrieved on 25 October 2017.

[21] Marcus Hegner. Methoden zur Evaluation von Software. 2003.

[22] Jan Hemming. Technologie und Produktion. In *Methoden der Erforschung populärer Musik*.

[23] Image-Line. FL Studio. https://www.image-line.com/flstudio/, retrieved on 10 January 2018.

[24] Cockos Incorporated. NINJAM. https://www.cockos.com/ninjam/, retrieved on 20 December 2017.

[25] Cockos Incorporated. REAPER. https://www.reaper.fm/, retrieved on 10 January 2018.

[26] Kompoz LLC. Kompoz Music Collaboration. https://www.kompoz.com/music/, retrieved on 20 December 2017.

[27] J Jenny Li, Tangqiu Li, Zongkai Lin, Aidtya Mathur, and Karama Kanoun. Computer supported cooperative work in software engineering. In *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*, pages 328–vol. IEEE, 2004.

[28] Jeffrey Lim. The Official Impulse Tracker Page. http://www.users.on.net/~jtlim/ImpulseTracker/, retrieved on 20 December 2017.

[29] John Maeda. *The laws of simplicity*. MIT press, 2006.

[30] Eduard Müller. Renoise. https://renoise.com/, retrieved on 10 January 2018.

[31] MuseScore BVBA. Sheet music at your fingertips. https://musescore.com/, retrieved on 20 December 2017.

[32] Chris Nash. *Supporting virtuosity and flow in computer music*. PhD thesis, University of Cambridge, 2012.

[33] Chris Nash. Manhattan: End-user programming for music. 2014.

[34] Chris Nash and Alan Blackwell. Flow of creative interaction with digital music notations. 2014.

[35] Simon Nestler, Eva Artinger, Tayfur Coskun, Yeliz Yildirim-Krannig, Sandy Schumann, Mareike Maehler, Fabian Wucholt, Stefan Strohschneider, and Gudrun Klinker. Assessing qualitative usability in life-threatening, timecritical and unstable situations, 10. In *Workshop Mobile Informationstechnologien in der Medizin (MoCoMed 2010)*, 2010.

[36] Nobuyuki. OpenMPT feature request: Networking. https://forum.openmpt.org/index.php?topic=3857.0, retrieved on 20 December 2017.

[37] Ohm Force SARL. ohmstudio.com, Ohm Studio website. https://www.ohmstudio.com/, retrieved on 20 December 2017.

[38] Markku Reunanen et al. Computer demos–what makes them tick. *Licentiate thesis, Helsinki: Aalto University*, 2010.

[39] Franca-Alexandra Rupprecht, Bernd Hamann, Christian Weidig, Jan C. Aurich, and Achim Ebert. IN2CO - A Visualization Framework for Intuitive Collaboration. In Enrico Bertini, Niklas Elmqvist, and Thomas Wischgoll, editors, *EuroVis 2016 - Short Papers*. The Eurographics Association, 2016.

[40] Franca-Alexandra Rupprecht, Taimur Kausar Khan, Gerrit van der Veer, and Achim Ebert. Criteria catalogue for collaborative environments. In *BISL*, 2017.

[41] Johannes Schultz. OpenMPT website. https://openmpt.org/, retrieved on 20 December 2017.

[42] Johannes Schultz. The Mod Archive, of the world's largest collections of music modules. https://modarchive.org/, retrieved on 2 January 2018.

[43] Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, pages 59–68. ACM, 1998.

[44] Oskari Tammelin. Jeskola Buzz. https://jeskola.net/buzz/, retrieved on 10 January 2018.

[45] Star Taylor. Schism Tracker. http://schismtracker.org/, retrieved on 10 January 2018.

[46] The Etherpad Foundation. Etherpad. http://etherpad.org/, retrieved on 3 January 2018.

[47] Hui Yu, Hongcheng Deng, Zhenxiang Zhao, and Yu Xiao. Transformation of landscape planning and design in computer supported cooperative work era. In *Computer Supported Cooperative Work in Design, 2008. CSCWD 2008. 12th International Conference on*, pages 1112–1115. IEEE, 2008.

Appendices

A User Experiment Tasks

Task 1: Connecting
Open the networking dialog by clicking the Connect icon. Connect to the server address `SERVER ADDRESS`. Note down the available documents and how many people can still join them:

Task 2: Communication

1. Connect to the shared song Task2 as a collaborator.
2. Make sure the chat dialog for the current document is open.
3. Write a chat message in the chat dialog belonging to the edited document.
4. Note down the names of all people that have participated in the chat so far.

Task 3: Concurrent editing
Let's edit the song from Task 2 together.

1. Create a new pattern and insert a note sequence of your choice. Note down the inserted pattern ID:
2. Convert all samples to instruments by inserting a new instrument in the instrument editor.

Task 4: Following collaborator changes
Collaboration mode adds an action log, found in the chat window, where you can find a condensed view of all user actions in the current session.

1. Observe the changes your collaborator makes to the document. Write down the pattern IDs and channel numbers in which edits are made:
2. Connect once again to `SERVER ADDRESS` and join the same shared song as a spectator.
3. Follow your collaborator by clicking on the nickname in the user list and switching to the pattern view. Note down the visited patterns, as well as the edited patterns:

Task 5: Rights management
How can you create a shared song with 1 collaborator and 3 spectators? Briefly describe the actions required to accomplish this:

Task 6: Concurrent editing

1. Connect once again to `SERVER ADDRESS` and join the shared song Task6 as a collaborator.
2. Find out which patterns have been locked by the collaborator. What happens when you edit them?
3. Lock pattern 3 and edit it.

Task 7: Annotations
Pattern annotations help organizing the collaboration and are not lost between editing sessions.

1. Add an annotation to your edited pattern data.
2. Find all annotations in the current song and write down their positions (pattern / channel):

Task 8: Combined Task 1

1. Connect to `SERVER ADDRESS` and join the shared song Task8 as a collaborator.
2. Edit any patterns you like, and try to keep track of your collaborator's actions as much as possible at the same time. Note down the collaborator's edited patterns at the end of this task:
3. Edit the volume envelope of instrument 14.
4. Edit sample 8.

Task 9: Combined Task 2

1. Connect to `SERVER ADDRESS` and join the shared song Task9 as a collaborator.
2. Create a new instrument and drag instrument 0 (Piano 1) from `GM.DLS`'s Melodic bank into it. Note down the instrument's number:
3. Note down the associated sample numbers:
4. Adjust the instrument's volume envelope.
5. Lock patterns 0 and 1.
6. Which instruments have been edited by your collaborator?

B Source Code

The source code is available both at https://gitlab.rhrk.uni-kl.de/j_schult/openmpt/tree/networking and on the CD submitted with the thesis. The CD contains both the full source tree including build instructions as well as a .patch file against the official *OpenMPT* repository (https://source.openmpt.org/svn/openmpt/trunk/), revision 9430.