INSTITUT FÜR INFORMATIK
der Ludwig-Maximilians-Universität München

A Collaborative Text Editor for the Learning Management System Backstage 2

Konrad Fischer

Bachelorarbeit

Aufgabensteller	Prof. Dr. François Bry
Betreuer	Prof. Dr. François Bry, Sebastian Mader
Abgabe am	30.10.2018

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Hilfsmittel verwendet habe.

München, den 30.10.2018 Konrad Fischer

Abstract

Verbal communication is often used for the transmission of solutions to tasks between students and from student to lecturer in group work in the university context. As verbal communication has several disadvantages, this thesis aims to provide a secondary mean of communication in the form of a collaborative editor. The proposed concept consists in the students working in groups using the collaborative editor and then digitally submitting their work to the lecturer. First, studies about the use of similar collaboration tools, focused on the tool *GoogleDocs*, are examined with the conclusion that the use of collaborative editors in education can have a positive effect on the learning experience and academic performance of the students. After that the concept is explained in more detail, is examined from a psychological point of view, and is found to contain elements of collaborative learning, social constructivism, the flipped classroom, and team-based learning. Furthermore the challenge of keeping several copies of the same text document in sync is presented, followed by different solutions to this challenge in the form of algorithms. From these algorithms the Logoot algorithm combined with an h-LSEQ allocation function is chosen for the exemplary implementation of the aforementioned concept in the form of a collaborative text editor embedded in the learning management system *Backstage 2*. Finally, the implementation is described in more detail and potential future additions to the concept are discussed.

Zusammenfassung

Verbale Kommunikation wird oft zur Übertragung von Lösungen für Aufgaben zwischen Studenten und von Student zu Dozent bei Gruppenarbeit im Universitäts-Kontext genutzt. Da verbale Kommunikation einige Nachteile hat, ist das Ziel dieser Arbeit ein sekundäres Kommunikationsmittel, in Form eines kollaborativen Editors, bereitzustellen. Das vorgestellte Konzept besteht darin, dass die Studenten in Gruppen unter Nutzung des kollaborativen Editors arbeiten und dann ihre Arbeit digital dem Dozenten zusenden. Zu Beginn werden Studien über die Nutzung von ähnlichen Kollaborations-Werkzeugen, fokussiert auf das Tool *Google Docs*, untersucht mit dem Ergebnis, dass die Nutzung von kollaborativen Editoren im Bildungs-Umfeld einen positiven Effekt auf die Lernerfahrung und akademischen Leistungen der Studenten haben kann. Anschließend wird das Konzept genauer erklärt, aus einem psychologischen Blickwinkel untersucht, und es wird festgestellt, dass es Elemente von Collaborative Learning, Social Constructivism, Flipped Classroom und Team-Based Learning enthält. Des Weiteren wird die Herausforderung des Synchronhaltens von mehreren Kopien des selben Dokuments vorgestellt, gefolgt von verschiedenen Lösungen für dieses Problem in der Form von Algorithmen. Aus diesen Algorithmen wird der Logoot-Algorithmus kombiniert mit der h-LSEQ Allokationsfunktion ausgewählt für eine beispielhafte Implementierung des erwähnten Konzeptes in Form eines kollaborativen Texteditors für das Learning Management System *Backstage* 2. Schlussendlich wird die Implementierung detaillierter beschrieben und mögliche Erweiterungen für das Konzept werden diskutiert.

Acknowledgements

Thanks to Prof. Dr. François Bry for the opportunity for this thesis and the helpful feedback during its making. Thanks to Sebastian Mader for the supervision and support of my work, the help when problems arose, and allowing me to work with the Learning Management System *Backstage 2* that he is the main developer of.

Contents

CHAPTER 1

Introduction

Group work in the context of tutorials, courses, or lectures at universities in many cases involves the production of an artifact, like for example a short essay. The collaborative editing of such an artifact can be challenging in the situation of most university lecture halls considering the room layout, as their classic layout consists of rows where the students sit next to each other making it hard for all group members to see the artifact that is being produced. This can lead to students trying to orally communicate their input to the solution to their group members, which, apart from being loud, has several other disadvantages. It can also lead to every group member producing an artifact on their own followed by a complex and time consuming merging process of the different solutions. Apart from that, oral communication is also used for the submission of the results of group work to the lecturer in the context of university courses.

One of the disadvantages of oral communication of solutions to sophisticated assignments is that the oral transmission of the produced artifact often takes very long, as the *sender* has to read or describe his whole solution and the *receiver* has to reproduce the artifact, for example by writing down a text, to have a visual representation of it that then can be processed further. This time needed for the conversion of the artifact into speech followed often by the reverse conversion back to the original artifact cannot be used to further refine the solution or to listen to instructions by the lecturer in the case of communication between group members. In the case of communication between a student and the lecturer, the time spent communicating solutions to tasks to the lecturer is time that could not only benefit the members of the group if used for other activities, like in the previous case, but all students taking part in the course.

Apart from that, the oral transmission of solutions is also error-prone, especially when small details are sent. An example of such details is the indentation of the lines of code of a program that could be the solution to a task in a computer science course. If the receiver of the information notices errors introduced into the artifact, they have to ask for a part or even the whole artifact to be transmitted again, which takes time that could be used for educationally more valuable activities. If the receiver does not notice the errors, they will further process the received information, which can lead to the correction of mistakes that the student did not make, if the receiver is the lecturer, or the further group work being based on a flawed artifact, if the receiver is another group member.

As a solution to this problem in the learning process a concept around a collaborative text editor is presented in this thesis to complement the oral communication in the groups

and between the students and the lecturer. Students use this editor on their laptops to solve assignments in group work. The process of collaboration is much easier and more efficient this way as all team members can always see the current state of the artifact being produced and edit it as the collaborative editor works in real time. Therefore, less oral transmission of information is necessary inside the group. When a group work phase of a tutorial is over or when a group has finished working on a task, the solution is sent to the lecturer, which eliminates the need for oral communication of solutions to the lecturer and therefore the presented disadvantages of this process.

The concept of collaborative group work presented in this thesis is not limited to texts as artifacts being produced in the process. The developed system can also be used for collaborating on different kinds of content that can be linearised (meaning that a total order exists on the content's data structure) like tables, for example in economics, or diagrams. Despite the different requirements concerning the management and synchronization of these types of content, they could be integrated into the system built in the frame of this thesis. An example of a special requirement of a content type would be the rule that all fields of a column of a table have to be filled before the column is sent to the other team members.

In this thesis related work with a focus on publications about the use of collaborative text editors in education will be examined first, then the proposed concept will be described in more detail and the psychological foundation of the concept will be explained. After that the challenge of managing multiple copies of a shared text document on clients will be described and solutions in the form of multiple classes of algorithms will be presented and evaluated. Finally the implementation of the presented concept will be outlined.

CHAPTER 2

Related Work

Collaborative writing tools have been the subject of many research papers by scientists from different fields. The best known collaborative word processing software is the suite *Google Docs*, *Google Sheets*, and *Google Slides*, a word processor, a spreadsheet and a presentation tool, respectively. The applications are part of a free, web-based software office suite offered since March 9, 2006 within the Google Drive service [1]. *Google Docs*[1] supports real-time editing by multiple users, simple sharing of documents via hyperlinks and includes, in its current version, a chat function that eliminates the need for a separate communication channel during collaboration [16].

Kessler, Bikowski and Boggs studied in [20] the effects of collaborative writing using *Google Docs* on second language learners at a university and found the collaboration in the student groups successful. The survey indicates the appreciation of the students for the collaborative writing using the online tool and a feeling of successful collaboration among team members [20].

Jeong evaluated in [18] the impact of using *Google Docs* on the collaboration of college students learning English as a foreign language. She found that most of the students enjoyed submitting their work via *Google Docs*, had in general positive attitudes about their use of *Google Docs* and saw in *Google Docs* a platform that could be useful in organizing their studies. 90% of the participating students enjoyed the experience of getting online feedback on their work. The concerns voiced in the survey included "reluctance in sharing their writing due to shyness or accidental missing data derived from periodic unstable situation of the online writing system" [18, p. 5].

Blau and Caspi examined in [8] the differences between sharing and collaborating on a written assignment using *Google Docs*. They found that students think that collaborative editing has a positive effect on the quality of their works. Students seemed to favor sharing of their document over editing and thought that their edits improved the quality of the documents of their peers while edits of other students of a student's own document reduced its quality. The authors also found that edits of a document resulted in lower levels of psychological ownership whereas publishing resulted in higher levels of ownership. These findings led to the authors' suggestion that a student's learning is better fostered by receiving feedback from other students than by them editing the student's document.

Zhou, Simpson and Domizi report in [52] on measuring the effectiveness of the use

[1] `https://www.google.de/intl/en/docs/about/`

of *Google Docs* for an out-of-class collaborative writing task by examining the influence of using *Google Docs* on the learning experience of the students. Their findings included that 95% of the students saw in *Google Docs* a useful tool for teamwork. While using the tool had no significant influence on the students' grades in the course, most students reported that *Google Docs* had positively influenced their group's collaborative experience.

Suwantarathip and Wichadee analysed in [43] the impact of *Google Docs* on the collaboration of students outside the classroom, in particular on their writing abilities. They found a significant difference between the scores of the students using *Google Docs* and those relying on traditional face-to-face collaboration, indicating a better performance of the users of *Google Docs*. Students reported positive attitudes toward collaborative writing and high collaboration in the groups using *Google Docs*. Most of the students found the application easy to use. The participants did not think that they learned much better but they nonetheless achieved better test results on the examination. In contradiction to the study conducted by Blau and Caspi the students did not seem to care for private ownership and were comfortable with their texts being deleted or edited by others.

Cardoso and Coutinho evaluated in [11] *Google Docs* in a different setting, a statistics module in vocational education. 95% of the students reported that they found using *Google Docs* pleasant. The participants did not experience major difficulties in using the tool. Furthermore all students found *Google Docs* to have a positive effect on the learning of mathematics and would use it again. Most experienced the work with the application as motivating and stimulating and considered it a useful tool for future work. Another finding was that *Google Docs* improved the students' opinion about both statistics and mathematics in general.

The results of the aforementioned studies have in common that the usage of *Google Docs* for learning has a positive influence on the students like a perception of an improved team work, a better quality of the results of group work and an increased collaboration. In some cases, using the tool even seemed to have a positive influence on students' grades. Furthermore, the aforementioned studies report a positive usage experience and more positive attitudes towards the collaboration process and towards the subject after using the web application. This leads to the conclusion that *GoogleDocs* has a positive influence on the learning process. Issues that the aforementioned studies have revealed include fear of data loss, shyness to share work, and a lower sense of psychological ownership but there are contradictory results on this last issue.

Google Docs is not the only real-time collaborative text editing application. Another well-known tool is *Etherpad* [44][2], which also allows for live editing of documents in the browser. Brodahl, Hansen and Hadjerrouit compared *Google Docs* to *Etherpad* in a study among education students [9, 10]. They were not able to verify that *Etherpad* users have a more positive attitude to the tool than to *Google Docs*. Only a minority of the students was motivated to use the collaborative editing tools and found that the use of the tools increased the quality of collaboration in the groups but 47% of the students enjoyed commenting on and editing other students work in their groups. Furthermore a majority of students reported that the tools did not work as they expected. Qualitative analysis revealed too large groups as a negative aspect and the ability to work from different places at different times as a general positive aspect. Most of the reports on editing other students work were positive, highlighting the advantage of students being able to correct mistakes of other students. The study concluded that *Google Docs* and *Etherpad* open up new possibilities for communication in collaborative writing.

Chu and Kennedy compared *Google Docs* to the wiki software *Mediawiki* which is, among others, used at *Wikipedia*. Students used both softwares for group projects and reported a positive experience with the tools in general. Furthermore, students found *Google Docs*

[2] `http://etherpad.org/`

easy to use with a user-friendly layout and while the majority of the participants rated both *Google Docs* and *MediaWiki* as useful knowledge management tools, *MediaWiki* was seen as more effective. [19]

Studies comparing *Google Docs*, *Etherpad*, and *MediaWiki* to one another confirmed the general positive influence of the usage of the tools in collaborative work that has been found for *Google Docs* for the other tools. Google's web application was described as easy to use. There seemed to be no difference in usage experience between *Google Docs* and *Etherpad* but a slightly more positive feedback on using *MediaWiki*. Concerns voiced include too large teams and unfulfilled expectations.

In summary, it can be said that the evaluation of *Google Docs* and similar tools for learning is very positive. The use of such tools seems to have a positive impact on students' working and learning experience. As the goal of this thesis is to develop a collaborative text editor for usage in an academic setting with, on a basic level, the same functionalities as *Google Docs* and *Etherpad*, that is, collaboratively adding and removing pieces of text from a document in real time, I hope to design a tool that will have a similar positive influence on the students' work experience.

CHAPTER 3

Concept

To achieve the goal of solving the problems presented in the introduction by complementing oral communication in group work scenarios, a concept around a collaborative text editor is introduced in this chapter. Furthermore an exemplary application of the concept to *Backstage 2* is described and the psychological foundations of the concept are examined.

3.1 Basic Concept

The first step in the proposed concept for enhancing teamwork is to divide the participants of a session, which could for example be a lecture or a tutorial, into teams. The lecturer specifies the size and number of the teams and the participants are free to join teams that are not yet full. Once the team creation is completed, the participants work on tasks with their teams using a collaborative text editor that allows them to edit a document together in real time, seeing a synchronized text. This tool enables collaboration in teams as all team members can always see the current state of the team's work and can contribute to it, even if the students are sitting side by side in a row in the typical fashion of university lecture halls. When a team has completed the task it was assigned or the time for working on the task is over, the teams submit their work, that is the document produced using the collaborative editor, to the lecturer. This digital submission process eliminates the need to orally communicate complicated artifacts like long logical formulas or long code. Now the teacher can compare the submitted solutions and pick individual ones to present so as to highlight common mistakes or ideal ways to solve the task. The elimination of the time consuming oral dictation of solutions improves the efficiency of the submission process, making the lecturer able to present more solutions than possible with the standard oral submission process. The group work efficiency is improved since students do not have to use a single piece of paper or laptop for drafting a solution or shout solutions to team members too far away to see their writing what is likely to lead to submissions of improved quality. Furthermore, organisational overhead is eliminated giving students more time to solve the tasks and the lecturer more time to provide assistance or feedback to students.

3.2 Exemplary Application of the Concept to Backstage 2

Backstage 2[1] is a learning management system that provides means to augment lectures, seminars, and courses with interactive elements to create an optimal learning experience for the students. The system includes, among other modules, a slide view that is synchronized with the presentation of the lecturer, so that the students can ask and answer questions about specific slides in real time, and quizzes that can be conducted during the lectures and evaluated by the lecturer in real time.

At the beginning of the tutorial, the students log into *Backstage 2* and join the tutorial there. The lecturer logs into the system as well, joins the tutorial, and creates multiple teams for the students. The students can see the list of teams and choose a team to join, ideally inviting the students sitting next to them into their team. Now the lecturer presents a task exemplary of a group of tasks and the solution to that exemplary task. This conveys the students a basic understanding of how to solve tasks of this kind. After that, the students work on the remaining tasks within their team using the collaborative text editor. Meanwhile the lecturer is available to answer questions on the tasks. The teams submit their solutions in *Backstage 2* when they are ready or when the work period is over. The lecturer can now see all the submitted solutions in the system and choose which ones to present using the projector.

3.3 Psychological Foundations of the Concept

The concept of students working together in small groups to solve a problem has a number of advantages. First, the proposed concept qualifies as an instance of collaborative learning, which "refers to an instruction method in which students at various performance levels work together in small groups toward a common goal. The students are responsible for one another's learning as well as their own. Thus, the success of one student helps other students to be successful" [15, p. 22]. Most works that examine collaborative learning are based on Lev Semyonovich Vygotsky's research, in which he emphasizes that collaborative learning is important in helping students to progress through their zone of proximal development, that is the space between what the student can achieve on his own and working together with more skilled partners [46].

There are claims that discussion of thoughts and ideas that takes place in small groups during group work, like in the proposed concept, leads to improved critical thinking and more interest in the lessons. There is evidence for these groups engaging in more sophisticated thought processes and being able to remember information longer than students working alone. There is also evidence for discussions in small teams contributing to make students more critical. Gokhale conducted a study [15] where he examined whether there is a significant difference in test scores beween students learning on their own and students working in groups. He found no significant difference on 'drill-and-practice' questions but significantly higher scores of the students that learned collaboratively on 'critical-thinking' questions. Students from the collaborative groups also indicated that the process helped to bring down their worries about problem-solving, was helpful in improving their understanding of the studied subjects and aided their cognitive process. [15]

On a team level, collaborative learning can lead to students questioning their own points of view as they are confronted with those of others which facilitates learning because students evaluate and change or stabilize their mental concepts of the topic. Furthermore, cooperation and the group that the students work in can provide encouragement and social support for individual members. The study in [5] suggests that collaborative learning

[1] `https://backstage2.pms.ifi.lmu.de:8080/`

can be even more effective with the help of computer technologies, which validates the proposed approach that uses a collaborative text editor. [5]

Apart from the classification as collaborative learning approach, the concept proposed in this thesis also contains elements of social constructivism. The term refers to the position that the creation of knowledge that takes place in a student "is the product of social interaction, interpretation and understanding" [3, p. 245]. An essential element of the social constructivist approach is therefore the emphasis on the learner instead of the information that is being transmitted and the view that no knowledge can exist without the meaning the learner gives to it from their experience within a group of peers. Based on this framework of social constructivism, a number of pedagogical theories have been proposed that share some principles, one of which being "View learners as active co-constructors of meaning and knowledge" [3, p. 247]. The roots of this recommendation are that learning is a social process and the interaction with others plays an essential role in the formation of knowledge. Interacting not only with a teacher but also with other students gives learners the opportunity to construct their own understanding of a topic and to self-reflect. These conversations are based on the theories and knowledge that the students have already acquired, for example in a preceding lecture or in former learning. One of the elements of the process of knowledge construction that these interactions are particularly useful for is the assessment of the state of knowledge of the other participants. The assessment makes it possible to identify topics that need further explanation. As the concept proposed in this thesis heavily relies on group work, the aforementioned processes, that are desirable for learning, can take place. [3]

Furthermore the concept proposed in this thesis is also connected to the teaching format of the flipped classroom. One definition of this format is "interactive group learning activities inside the classroom, and direct computer-based individual instruction outside the classroom" [7, p. 5]. Therefore, the proposed approach contains the group learning component of the flipped classroom concept. Research on the usage of the idea in practice generally concluded that students' opinions were positive but there were always some students that did not like the pedagogical change [7]. More specifically studies evaluating the flipped classroom concept in university environments found that students experienced the new course design as more effective [51], became "more open to cooperative learning and innovative teaching methods" [40, p. 171], prefered the interactive learning phases over traditional lectures by a teacher [45]. Apart from that students that took part in flipped lessons were found to score higher in exams in [36].

The problem the flipped classroom approach tries to tackle according to [2] is the passive role of students in lectures, which is one of the issues the concept proposed in this thesis is also focused on. The flipped classroom identifies the solution as active learning which is defined as the students solving problems to help them understand a concept instead of listening to a traditional lecture. One way of providing a theoretical background for the flipped classroom concept is Self-Determination theory, which defines three basic cognitive needs namely competence, autonomy and relatedness, whose satisfaction should improve students' motivation. For increasing the intrinsic motivation of a student, a feeling of relatedness and security is needed in addtion to competence and autonomy. The former two needs are fulfilled when students actively work on assignments in groups as they feel more connected to the teacher and their peers this way leading to an increase in intrinsic motivation. The flipped classroom approach also aims to improve the extrinsic motivation of learners. Research found that students, who actively participate in the construction and spreading of knowledge, perceive themselves as more competent than students listening to conventional lectures. By satisfying the students' need for competence, the flipped classroom is likely to generate extrinsic motivation in students. Furthermore, through facilitating students' active involvements in lessons, the flipped classroom creates a space that makes the formation of learning groups more likely, thereby improving the relatedness the

students feel which again leads to an increase in external motivation. As the concept proposed in this thesis utilizes active learning, it should improve the intrinsic and extrinsic motivation of learners. [2]

Another educational technique that the concept proposed in this thesis contains elements of is team-based learning. It is made up of four elements, namely "[s]trategically formed, permanent teams (...) [r]eadiness assurance (...) [a]pplication activities that promote both critical thinking and team development (...) [, and] [p]eer evaluation" [25, p. 41] and aims to change traditional lessons into a space where students assist each other in learning how to apply learned concepts in practice. The concept proposed in this paper implements the activities component of team-based learning. The authors of [25, 24] claim that team-based learning has several advantages for students, one of which is that because the approach utilizes real teams and not just temporary working groups or individual students, the teacher can assign more difficult tasks to the teams, leading to students learning more by working on harder problems that they could not solve on their own. Although the concept involving the collaborative text editor does not utilize permanent teams it could be extended to do so. Temporary working groups are still able to tackle more difficult problems than individual students are on their own and therefore more efficient learning occurs than in traditional lectures. Furthermore, a peer evaluation component is not currently included in the proposed concept but could be easily added in the future as a step in the process. It could take place before the teacher presents several submitted solutions and therefore give students the opportunity to compare their perceptions about the quality of other solutions with the evaluation of these by the teacher. Another way of implementing peer evaluation would be to add it to the end of the process after the teacher presented the submitted solutions to provide every student with an evaluation of their solution, even if it was not reviewed by the teacher in front of the whole course. Another advantage of team-based learning that the authors of [24] claim is that all students learn to appreciate the value of working in teams in order to find solutions for difficult problems. Also team-based learning is a remedy for the low participation levels in traditional lectures because it encourages students to be a part of the course in a more active way, what has a positive effect on both the learner and the instructor [24]. Team-based learning has been proven to have a positive influence on learning in a number of studies. Effects include improved test scores [22], more participation [38], students being more pleased with their learning experience [6, 50], and groups as a whole performing better than their best members on their own [47].

CHAPTER 4

Collaborative Editing

In this chapter problems arising in collaborative text editing are discussed followed by the presentation of an evaluation of a number of algorithms that represent solutions to those problems. This evaluation motivates the choice of the synchronization algorithm used to implement the proposed concept of groupwork using a collaborative text editor.

4.1 Simultaneous Editing of a Common Document

Editing a text document can be broken down to two basic operations, `Insert(c, i)` and `Delete(i)` where `c` is a character and `i` is an index in the string that represents the document. As the goal of collaborative editing is to enable multiple users to work on the same document at the same time, these operations need to be scaled to multiple clients. A naive way to do this is just sending every operation that is executed on a copy of the document to all other clients, which then re-execute the operation. This works well when the latency in the network is low, for example in a fast local area network (LAN) as depicted in Figure 4.1 where the horizontal arrows represent a nearly instant delivery of the messages in the network. Problems arise in networks with significant latencies as they can lead to the operations being executed in different orders on different clients. This different execution order can then result in different states of the document on different clients, although all operations were executed on all clients. [21, 17]

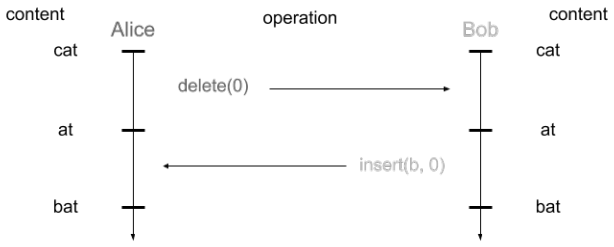

Figure 4.1: The naive approach to scaling text editing (based on [21])

An example of such a synchronization problem is presented in Figure 4.2. Both copies

start with the same string `cat`. Then Alice executes the operation `delete(0)` to delete the first character of the text on her copy of the document and this operation is sent to Bob. Concurrently, Bob performs `insert(b,0)` to insert the character `b` at the beginning of the string and his operation is sent to Alice. When Bob's `insert` operation is received by Alice it is executed and produces the expected result `bat`. Bob receives Alice's `delete` operation and applies it but here the operation does not have the intended effect (to delete the character `c`) as Bob's `insert` operation has been executed before. The operation produces the result `cat` and now the system is in an inconsistent state as all operations have been executed on all clients but the clients are in different states, or in other words there is a divergence between the clients. If a serialization protocol is used to ensure that all clients execute all operations in the same order, for example Bob's `insert` operation and then Alice's `delete` operation, the same result `cat` would be obtained on both clients in that case but that result would not include the intended effect of Alice's operation as the character `c` is still present in the result. [17]

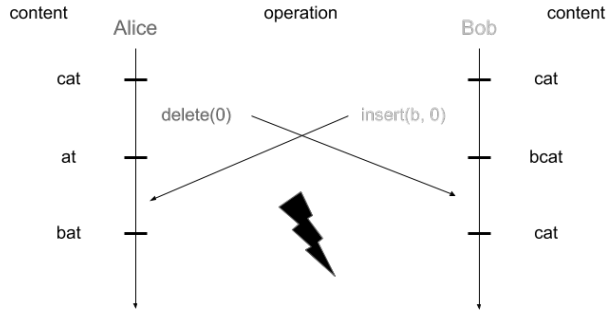

Figure 4.2: An example of synchronization issues in a network with latency (based on [17])

4.2 Synchronization Algorithms Using Optimistic Replication

Before examining different approaches to synchronization, a few terms have to be defined. "Any replicated system has a concept of the minimal unit of replication. We call such unit an object. A replica is a copy of an object stored in a site, or a computer. A site may store replicas of multiple objects, but we often use terms replica and site interchangeably, since [many] algorithms manage each object independently" [37, p. 2].

There are two general approaches to the replication of data. The established family of replication strategies is called pessimistic replication. It tries to create the impression of editing a single document for the user. There are different ways to achieve this effect but most of them share a common concept consisting in the prevention of access to replicas of the data that are not guaranteed to be up to date. Replicas are managed synchronously during actions and users are not allowed to edit a document during an ongoing update by another user, as this is the trivial case of a replica not being up to date. Pessimistic replication delivers good results when there is very little latency and errors only occur rarely but in environments like the Internet algorithms based on pessimistic replication perform poorly and have limited availability. [37]

Optimistic replication is a more recent approach but is already used in many services, for example in the Domain Name System (DNS) on the Internet. It has a fundamentally different approach to concurrent editing as users are allowed to read or write data without the

guarantee that their replica is up to date. The name of this family of algorithms originates from the optimistic character of its basic assumption that synchronization issues will only occur rarely. In algorithms using optimistic replication conflicts are not prevented a priori but fixed when they occur and operations executed on one replica are immediately sent to the others in the background. Optimistic replication has a number of advantages over the pessimistic approach. The most important advantage is that a document synchronized by means of optimistic replication can be asynchronously edited by the participants meaning that multiple users can edit the document simultaneously without having to wait for other users to finish their actions first. Furthermore, documents can be updated instantaneously with the effect of local actions as those updates are applied preliminarily. This also leads to documents still being editable, although without receiving updates from the other replicas, during network outages. Moreover, applications using optimistic replication potentially scale to a larger number of clients as less synchronization over the network is necessary because some synchronization mechanisms, like the one that ensures that only one replica is edited at a time and would be needed for pessimistic replication, can be dropped. [37]

Improving availability in a distributed system often comes at the cost of consistency. Algorithms using a optimistic replication approach have to deal with diverging copies of the document when a user edits locally and the changes have not been transmitted to the other replicas and executed there yet, whereas documents that are synchronized in a pessimistic way are always in sync. Moreover, in optimistic systems conflicts between concurrent actions can arise which are prevented a priori in their pessimistic counterparts. Because of these drawbacks, optimistic replication algorithms can only be used in scenarios where conflicts occuring and copies of the document having different states for limited amounts of time are acceptable. [37]

An optimistic replication algorithm applies updates the user makes to the document instantaneously locally and then sends them to the other replicas while always executing operations received from them in the background. Due to this procedure, optimistic replication can only guarantee that the different copies of the document will converge eventually as they can diverge temporarily. This guarantee is called eventual consistency as contrasted with the 'normal' consistency that pessimistic approaches guarantee. [37]

In the form of synchronization algorithms an array of different concrete solutions for the problem of keeping multiple copies of a document in sync exists. The best known approaches are Operational Transformation (OT), Differential Synchronization, and commutative replicated data types (CRDTs) [39].

4.2.1 Operational Transformation

As discussed before, the transmission of the operations on a collaborative document via the network can result in them arriving at different sites in different orders, possibly leading to a divergence in the document states. The Operational Transformation approach uses the concept of optimistic replication with its basic functioning being that operations that have been executed on a state of the document on one site are transformed so that they can be applied on the state of a different site resulting in the convergence of the replicas. [13, 34]

Figure 4.3 shows an example of an operation being transformed. Both participants start with the same string `cat` being the content of their document. Alice decides to delete the first character by executing the operation `delete(0)` and Bob concurrently inserts a `b` at the beginning of the string by executing `insert(b,0)`. Alice's and Bob's operations are both applied to their local replicas and then sent to the other site. Bob's operation arrives at Alice's replica and is transformed. The transformed operation is the same as the original operation Bob sent as his intention to insert the character `b` at the beginning of the string is preserved when the operation is left unchanged. When Alice's operation is received by Bob's replica, the system detects that the operation has to be transformed as its intention,

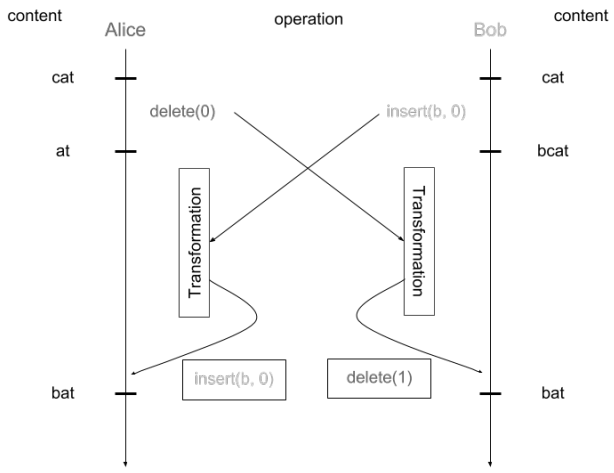

Figure 4.3: An example of an operation being transformed (based on [12])

deleting the character `c`, would be lost if it would be applied unchanged. The character `c` is at the position `1` in Bob's copy of the document so the operation is transformed to `delete(1)` and the transformed operation is applied.

The process of sending and receiving an operation using Operational Transformation starts with the generation of the operation at the sender site where it is executed and then, supplemented by metadata, sent to the other sites. When a site receives the operation, the metadata is examined to determine whether the sending site has executed operations the receiving site has not executed yet. If this is the case, the operation is queued. If not, the receiving site checks if it has executed operations that the sending site has not executed yet. If so, the operation is transformed using the attached metadata before being applied. [13]

A module of a collaborative editor tasked with Operational Transformation usually consists of two parts: a transformation control algorithm, which is responsible for choosing the operations that should be transformed against other operations based on their concurrency relationships, and transformation functions that execute the transformations of operations based on various parameters. [42]

An evaluation of the speeds of algorithms for real-time collaboration found that the most representative OT algorithms that do not rely on a central server, *GOTO* and *SOCT2*, perform significantly worse than representative CRDT algorithms. The algorithms using CRDTs were between 25 and 1000 times faster on average in experiments. These results are consistent with the average time-complexities of the two OT algorithms [4]. Apart from their performance, another concern about OT-based algorithms is that most of them use vector clocks to determine a causal ordering between operations [48, 49, 31]. A vector clock in a system of n processes is an n-dimensional vector v of non-negative integers that each process p_i holds. The value $v[j]$ represents the logical time at process p_j, therefore the whole vector is p_i's view of the logical global time. Logical time progresses as each process p_j increments $v[j]$ by d (where $d > 0$) before executing an action. Each message transmitted in the system is appended with the vector clock v of the sender [33]. The usage of vector clocks in an algorithm for collaborative editing leads to scaling issues as the size of the vector v is proportional to the number of participants, leading to more data having to be sent over the network [48, 49, 31]. Another difficulty arising in peer-to-peer networks is that a client might not know all other clients and therefore be unable to produce a valid vector clock [31]. *MOT2* is an OT algorithm that does not require vector clocks but assumes that the transformation functions used in the OT process satisfy Transformation Property 1

(TP1) and Transformation Property 2 (TP2) as the fulfillment of these properties guarantees eventual consistency for certain OT algorithms [48, 35]. According to [35, p. 364] and [34, p. 290, 292] the two properties consist in:

- The transformation function *tf* yields a transformed version O_3 of an operation O_2 against an operation O_1: $tf(O_2, O_1) = O_3$
- TP1: For two operations O_1 and O_2 that have been executed concurrently on the same document state, *tf* satisfies TP1 if and only if first executing O_1 and then the transformed operation $tf(O_2, O_1)$ has the same effect as first executing O_2 and then the transformed operation $tf(O_1, O_2)$. TP1 represents a commutativity-like property of the transformation function.
- TP2: For three operations O_1, O_2, and O_3, where the pairs O_1 and O_2, O_2 and O_3, and O_1 and O_3 have each been executed concurrently on the same document state, *tf* satisfies TP2 if and only if "$tf(tf(O_3, O_1), tf(O_2, O_1)) = tf(tf(O_3, O_2), tf(O_1, O_2))$" [35, p. 364]. T2 ensures that the transformation of operations along different but equivalent paths leads to the same operation as a result. Figure 4.4 shows an example of two paths $tf_1(tf_1(r, r_2), r_1')$ and $tf_1(tf_1(r, r_1), r_2')$ that lead to the same operation with tf_1 being the transformation function.

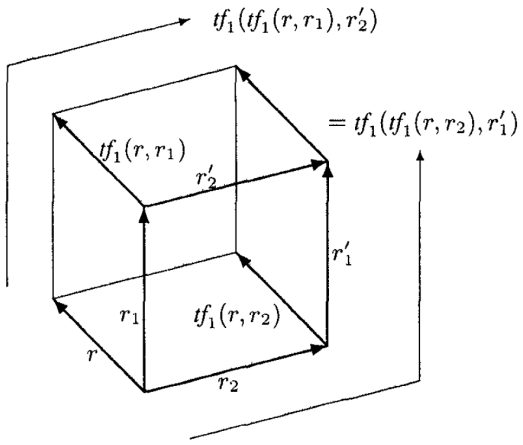

Figure 4.4: Consistent transformations of an operation *r* showing TP2 [34, p. 291]

According to [48] the only transformation functions that are both usable for text documents and satisfy these properties are *Tombstone Transformation Functions*. These functions are based on tombstones, meaning that deleted artifacts, like characters, are not removed from the data structure representing the document but replaced by tombstones [48]. Tombstones represent a significant overhead that degrades performance over time and their removal requires acquiring consensus between the clients in a network, which is an expensive operation [4, 28]. One approach requires a vector clock to track the state of other clients for save tombstone removal and another tries to make obtaining a consensus easier but in doing so limits the structure of the network and utilizes an expensive additional algorithm [28]. Another general issue with the OT family of algorithms is that the approach seems to be complex and error-prone, as established by Oster, Urso, Molli, and Imine that found fundamental errors in a number of popular OT algorithms in [30].

In conclusion, the Operational Transformation class of algorithms is not an ideal choice for an algorithm for a collaborative editing system as it has performance issues related to

the usage of vector clocks or tombstones and algorithms of this class seem to be overly complex.

4.2.2 Differential Synchronization

Differential Synchronization (DS) by Neil Fraser is an optimistic replication algorithm that uses deltas to send as little data over the network as possible and is convergent, meaning that errors, e.g. in applying the changes of one client at another client, do not cause different copies of the document to diverge. It is also well-suited for unreliable networks and asynchronous, meaning that the editor the user works in does not have to be locked because of the algorithm's operations at any time. The algorithm is suitable for any type of content, given *diff* and *patch* algorithms are defined for that type of data. Changes are detected by computing the difference between the current and the previous content of the client resulting in a *diff* describing them. This *diff* is then sent to all other clients. [14]

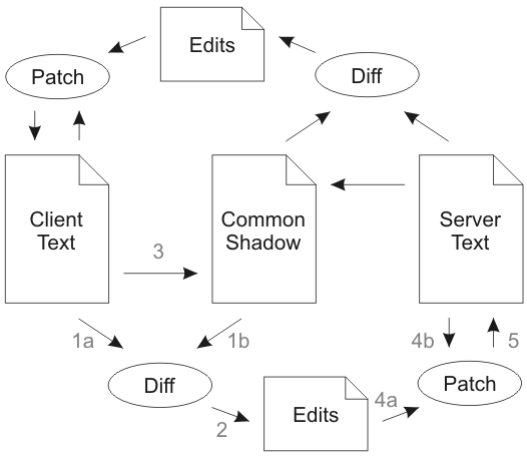

Figure 4.5: An iteration of the basic DS algorithm [14, p. 2]

DS uses a never ending loop of background *diff* and *patch* operations. In its most basic form, as depicted in Figure 4.5, that takes place on one computer between the two documents *Client Text* and *Server Text*, it starts with *Client Text*, *Common Shadow* and *Server Text* all being equal [14]. One iteration of the loop consists of the following steps [14, p. 3]:

1. The difference between *Client Text* and *Common Shadow* is computed.
2. The first step yields a list of changes that have been made to the *Client Text*.
3. *Common Shadow* is replaced by *Client Text*.
4. The modifications are adopted on a best-effort basis, resulting in a *patch*, a 'manual' describing how to apply the changes.
5. The patch is applied to *Server Text*.

The rest of the iteration consists in the same process being executed in a mirrored way with the difference that *Common Shadow* is now equal to the value that *Client Text* had in the first half of the iteration. This leads to the *diff* containing changes to *Server Text*. An important feature of DS is that the patch procedure is *fuzzy*. This means that patches are applied even if the data has changed between the time the patch was calculated and its

application time. If a subset of the patch can't be calculated in step 4, it will show up negatively in the next iteration's *diff* and will be removed from *Client Text*. [14]

The simple variant of the DS algorithm shown in Figure 4.5 is not suited for networks with multiple systems and is only intended for showing the functioning of the algorithm, despite the misleading naming of the components that Fraser admits to in [14]. In order to adapt it to a client-server scenario the shadow has to be split into two shadows that are updated independently: *Server Shadow* and *Client Shadow*. The concept of the algorithm remains the same as depicted in Figure 4.6. [14]

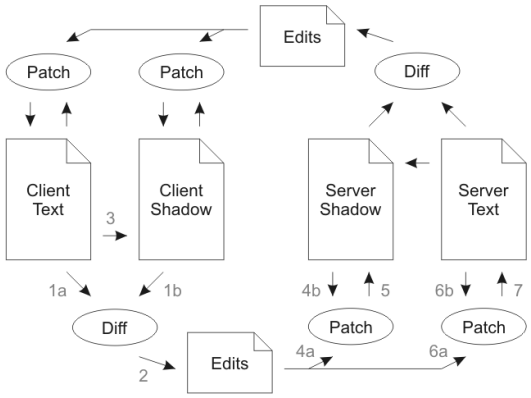

Figure 4.6: An iteration of the DS algorithm on two systems [14, p. 3]

Apart from the algorithm itself, Fraser also lists extensions of it addressing issues that can arise in networks. One of these extensions is the *guaranteed delivery method*, that queues edits when packages are lost and resends them until receiving an acknowledgement. It adresses data corruption, duplicate packages, lost outbound packages, lost return packages, and out of order packages. [14]

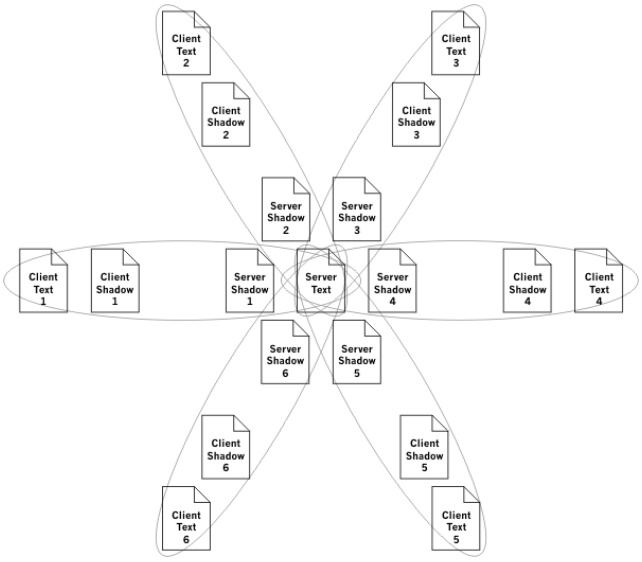

Figure 4.7: The layout of a multi-client network using DS [14, p. 5]

The layout used for communication between two parties in Figure 4.6 can be scaled to an arbitrary number of clients using a single server as depicted in Figure 4.7. Every oval

represents an instance of the topology in Figure 4.6. The *Server Text* is shared between all instances but each client requires its own *Server Shadow*. [14]

One issue of DS that Fraser mentions in his paper himself is that *diff* and *patch* are computationally expensive operations. Although this problem can be mitigated using heuristics to simplify *diff* operations and utilizing efficient *patch* algorithms, DS still puts a significant amount of stress on the server [14]. Apart from the computational workload, memory on the server could also be an issue when handling large documents with a large number of clients as the server needs a separate *Server Shadow* for each client. Another problem consists in the continuous execution of the algorithm, even if no changes are made to the document. Fraser mentions a method to adapt the synchronization frequency to the user's editing behavior but the fundamental problem persists [14]. Apart from these performance issues, there is also the problem of patch errors, meaning cases where the patch algorithm is not able to incorporate the changes of another party into a user's document [14]. Fraser recommends either dropping the problematic patches or asking the user to manually resolve the conflict. Neither of these alternatives are satisfying from a usability point of view. Dropping the patch is the way Fraser's implementation of DS chooses, as this describes the mentioned behavior of changes that cannot be applied showing up negatively in the next *diff* and being removed from the other party's document. Furthermore, the *guaranteed delivery method* represents an additional safety measure but should not be necessary in networks using *TCP* as this package transmission protocol resolves the issues Fraser's extension of the algorithm is supposed to fix.

In conclusion, the DS algorithm is not an ideal choice for collaborative editing as it has performance and usability issues.

4.2.3 Commutative Replicated Data Types

CRDTs are a general approach to collaborative editing. A document exists in the form of several copies on different clients (replicas) that can be modified independently. After a user modifies the document, the state of the replicas diverges. Local modifications made by users are then re-executed at the other replicas, and ultimately every modification is executed at every replica. Certain properties of the operations in a CRDT ensure that after executing all actions all replicas are in the same state [39]. To formally define these properties a few crucial concepts have to be defined first:

- We say that an operation α *happened before* an operation β if a client executes β after the same client has executed α [39].
- "Operations are *concurrent* if neither *happens before* the other" [39, p. 5, emphasis added].
- "Two operations α, β *commute* if, for any state T, execution sequences $\langle T \cdot \alpha \cdot \beta \rangle$ and $\langle T \cdot \beta \cdot \alpha \rangle$ are correct states and are equivalent" [39, p. 5, emphasis added].

A CRDT is defined as a data type with the property that all concurrent operations defined for it commute with each other. If operations are replayed in an order respecting the *happened before* relation at all sites, the state of all replicas is identical after executing all operations. Therefore, CRDTs ensure eventual consistency, as proven by Shapiro and Preguiça in [39]. No additional concurrency control measures, like the transformation of operations with OT, are necessary [32]. A history of operations or concurrency detection is not needed either because CRDTs are designed to be commutative [4].

Each CRDT document consists in a linear sequence of atoms. An atom can be a line of text, a character, or any other immutable value [39]. Users can modify a replica of the document by executing one of the following operations [39, p. 6]:

- `insert(newPos, newAtom)` inserts the atom `newAtom` at the position `newPos`. Atoms with positions smaller than `newPos` lie left of `newAtom` and atoms with positions greater than `newPos` lie right of `newAtom`. The ordering of the positions is determined by the ordering of the position identifiers.
- `delete(oldPos)` deletes the atom at position `oldPos`.

The position identifiers have the following properties [39, p. 7]:

- Each position has a unique position identifier.
- A total order $<$ is defined on the position identifiers.
- It is always possible to insert an atom between two existing atoms, as given two identifiers X, Z a new unique identifier Y can be generated so that $X < Y < Z$.

Figure 4.8 shows Alice and Bob editing a document using a generic CRDT. The bold numbers in the top row of the tables represent unique identifiers for the position of the characters beneath them. There is a total order on the position identifiers, in this example represented by the $<$ relation. The exclusive use of even numbers as identifiers in the example represents the possibility to always generate a position identifier between two existing position identifiers in a CRDT. Alice and Bob start the editing session with the same document state. Alice then deletes the character `c` at the unique position `2` with the operation `delete(2)`. Bob concurrently decides to add the character `b` to the beginning of the text. The CRDT generates a position at the start of the document, left of the character `c` with the position `2`, which is `0` in this case. The operation to execute Bob's desired action then is `insert(b,0)`. Both participants locally apply their operations and then send them to the other users in the editing session, according to the principle of optimistic replication. When Bob's operation arrives at Alice's replica, the `b` is positioned in the document string according to the total ordering of the position identifiers. As $0 < 4 < 6$ the new character is added to the beginning of the string. Upon arrival at Bob's replica Alice's delete operation causes the deletion of the character `c`, as intended by Alice, although a different character is at the beginning of the string now. This is possible because Alice's operation addressed the character at the unique position `2` and not a position that a new character could have taken by the time the operation arrives at Bob's replica, like a conventional index of a character in a string.

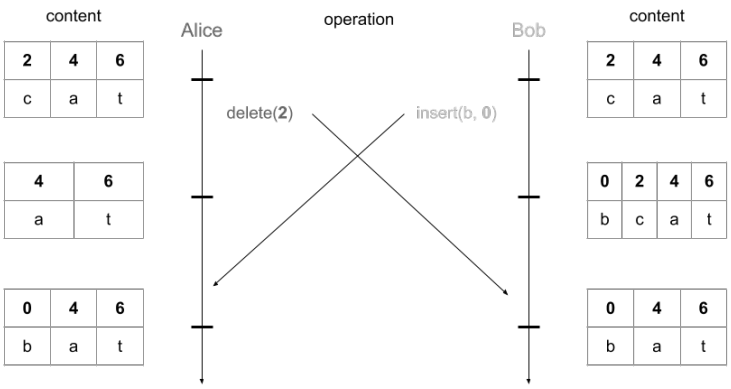

Figure 4.8: Collaborative text editing using a CRDT (based on [12])

4.2.4 Logoot

Logoot is a CRDT algorithm for collaborative editing systems with good scalability regarding the number of users and edits. It works without tombstones, has a logarithmic complexity relative to the document size, and meets convergence, causality-perservation and intention-preservation (CCI) criteria. The absence of tombstones leads to the space overhead being linear relative to the size of the document and no requirement for a garbage collector. [48]

CCI criteria have been found to be necessary for a working collaborative editing system [48, p. 3][41, p. 62]:

- Convergence: All replicas are in the same state when all operations have been executed at all sites. This includes eventual consistency.
- Causality-preservation: Operations are executed in order according to the *happened before* relation at all sites.
- Intention-preservation: The effect of executing an operation is the same at all sites and its effect does not interfere with the effect of other operations.

The authors of the Logoot algorithm also claim in [48] that it meets the criterion of numerical scalability meaning that it is able to handle new users or elements without significant performance degradation.

As Logoot is a CRDT implementing optimistic replication, its basic functionality consists in the delivery of the modifications a user has made on their replica to all other replicas where they are re-played leading to a potential temporarily difference between the copies. The available operations are `insert` and `delete`. Like in other CRDTs, Logoot operations commute which, together with a causal ordering, guarantees convergence. In Logoot's proposal for a total order between the position identifiers of elements, as required by the CRDT framework, each is based on a list of integers. [48]

A Logoot document is composed of elements, which are called lines in the original paper but could also be characters or whole paragraphs depending on the desired granularity. Two empty virtual elements are always present at the beginning and the end of the document to ensure that every insert operation consists in inserting a new element between two existing elements, even if the document is empty or the new element should be positioned at the beginning or the end of the document [48]. The definitions needed are the following [48, p. 4]:

- An element is a couple $\langle id, data \rangle$ where id is a unique position identifier and $data$ is the content of the element, for example a character.
- A position identifier is a tuple $\langle pos, c_s \rangle$ where $pos = [id_1, id_2, ..., \langle pos_n, sid_n \rangle]$ is a position and c_s is the value of the logical clock of the site s. Each position identifier is unique because the site identifier sid_n is unique. The existence of a logical clock at each site that is incremented when an element is created is assumed by the algorithm.
- "An identifier is a couple $\langle pos, site \rangle$ where pos is an integer and $site$ a site identifier" [48, p. 4].
- "A position is a list of identifiers" [48, p. 4].

A document can also be understood as a tree where each position represents a path from the root to a leaf [28]. The document `axbe` is for example represented by the tree in Figure 4.9.

The total order between position identifiers required by the CRDT framework is defined on Logoot's equivalent positions, as Logoot's position identifiers are ordered by their positions [48, p. 4]:

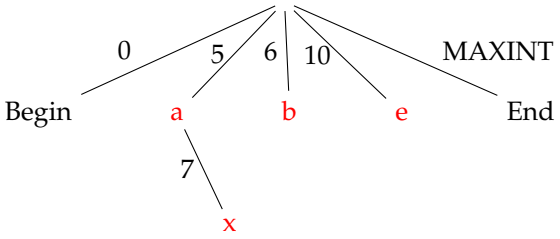

Figure 4.9: One possible tree representation of the Logoot document `axbe` (based on [28])

- Let $p = [p_1, p_2, ..., p_n]$ and $q = [q_1, q_2, ..., q_m]$ be two positions. "$p \prec q$ if and only if $\exists j \leq m : (\forall i < j : p_i = q_i) \wedge (j = n+1 \vee p_j <_{id} q_j)$" [48, p. 4]. This equals a lexicographic comparison of p and q.
- "Let $id_1 = \langle pos_1, site_1 \rangle$ and $id_2 = \langle pos_2, site_2 \rangle$ be two identifiers, we get $[id_1 <_{id} id_2]$ if and only if $pos_1 < pos_2$ or if $pos_1 = pos_2$ and $site_1 < site_2$" [48, p. 4]. Therefore $id_1 <_{id} id_2$ if and only if id_1's integer is smaller than id_2's integer or their integers are equal and id_1's site identifier is smaller than id_2's.

Deleting an element from a Logoot document is relatively simple: A replica generates a deletion operation for the desired position and sends it to the other replicas that then delete the element. Inserting an element into a document is more complicated as it implies the generation of a new position identifier. Because of the mentioned empty virtual elements or lines that always exist at the beginning and the end of the document, the case of inserting an element between two existing elements is the only case to cover. If the element $\langle id_x, data_x \rangle$ is inserted between $p = \langle id_p, data_p \rangle$ and $q = \langle id_q, data_q \rangle$, the new position identifier id_x has to satisfy $id_p < id_x < id_q$ and therefore the position of id_x has to satisfy an equivalent condition relative to the positions of id_p and id_q. The authors of Logoot define an algorithm that generates positions for a number of adjacent lines, as they argue that edits on a document often insert a series of lines. The algorithm can be found in Figure 4.10, where N is the number of inserted lines, p and q the positions between which the lines are inserted, and $MAXINT$ the base of the integers used in the generated positions [48]. The functions used in the algorithm are [48, p. 4]:

- `prefix(p,i)` returns an integer where each digit is the integer of the j'th identifier in the position `p` for $j \leq i$. If the position p has less than i identifiers ($|p| < i$), the result is filled with zeroes. The returned integer is in base $MAXINT$.
- `constructPosition(r,p,q,s)` returns a position $[\langle int_1, sid_1 \rangle, \langle int_2, sid_2 \rangle, ..., \langle int_n, sid_n \rangle]$ where int_i is the i'th digit of the number `r`. The effect of this function is that it generates a new position where each identifier's integer comes from the identifier at the same position in r. Each site identifier comes from the identifier at the same position in either p or q or is set to the value s. The source of the site identifier sid_i is determined according to a set of rules:
  1. if $i = n$ then $sid_i = s$
  2. if $int_i = p_i.int$ then $sid_i = p_i.sid$
  3. if $int_i = q_i.pos$ then $sid_i = q_i.sid$
  4. else $sid_i = s$

In order to obtain the shortest possible positions the algorithm in Figure 4.10 first finds the length of the shortest prefixes of p and q that have the same length and have space for at

```
1  function generateLinePositions(p, q, N, s){
2    list := {};
3    index := 0;
4    interval := 0;
5
6    while(interval < N){
7      index++;
8      interval := prefix(q, index) - prefix(p, index);
9    }
10   step := interval/N;
11   r := prefix(p,index);
12   for (j := 1 to N){
13     list.add(constructPosition(r + Random(1, step), p, q, s);
14     r := r + step;
15   }
16   return list;
17 }
```

Figure 4.10: Logoot's algorithm for generating new positions between two existing ones [48, p. 4]

least N positions between them in lines 6-9, filling p or q with zeroes if one of them or both are too short. The length of the prefixes is saved in `index`. Then the maximum distance between two of the N positions is calculated in line 11. After that in lines 12-16 N identifiers are generated randomly with a random distance $d \in [1, step]$ between each identifier and the next. The desired positions are generated by adding each identifier to the common prefix identified in the previous step which yields the list of new positions. The identifiers are allocated from left to right between p and q. The three steps of generating the new identifier, constructing the new position, and adding it to the result all happen in line 13 of Figure 4.10. In the tree representation of the positions in Figure 4.9 the two steps of finding the shortest common prefix and generating the new positions correspond to finding the highest possible level in the tree between p and q where at least N nodes can fit in between. Then that level and the previous higher levels are filled with nodes that lie between p and q. The distance d between two newly generated nodes is random but limited by $d \in [1, step]$. [48]

As an example on a site with site identifier sid_{new} a number of N new positions between the positions $p = \langle 1, sid_1 \rangle$ and $q = \langle 6, sid_2 \rangle \langle 17, sid_3 \rangle$ would be generated in the Set 4.1 if $N < 5$ or in the Set 4.2 if $N \geq 5$ [48, p. 4-5].

$$\{\langle i, sid_{new} \rangle \mid i \in]1, 6[\} \tag{4.1}$$

$$\begin{aligned} &\{\langle 1, sid_1 \rangle \langle i, sid_{new} \rangle \mid i \in [0, MAXINT]\} \cup \\ &\{\langle j, sid_{new} \rangle \langle i, sid_{new} \rangle \mid i \in [0, MAXINT], j \in]1, 6[\} \cup \\ &\{\langle 6, sid_2 \rangle \langle i, sid_{new} \rangle \mid i \in [0, 17[\} \end{aligned} \tag{4.2}$$

A random allocation function is used for the identifier generation to prevent different replicas from concurrently generating the same integers for the newly generated identifier that will be added to the end of the common prefix and form a new position in line 13 of Figure 4.10. This is not a problem itself as there is still an ordering between the two concurrently generated positions as the newly generated identifiers can be ordered by their unique site identifiers and therefore the positions can be ordered. The problem arises when

one of the two sites that concurrently generated the same integer wants to insert a new element between the previously generated positions as a longer position, or a corresponding new level in the tree representation of the document, has to be created. This could be necessary because there are no free integers between the previously generated ones and an ordering by site identifier is no option either as both are already in use for that integer. The longer positions increase the overhead of the algorithm and therefore it aims to prevent the concurrent generation of the same integers. [48]

Line deletion or insertion operations from other replicas can be applied in logarithmic time relative to the number of lines as the approach uses binary search to find the position where the line should be inserted or removed. The removal of lines from a document is safe as the total order between the remaining lines is not altered. Positions of removed lines can be reused. [48]

The authors of the Logoot algorithm conducted a series of experiments on modifications of *Wikipedia* pages and found that despite the theoretically infinite length of the positions for each element the algorithm can work effectively in practice. Logoot's overhead was lower than those of tombstone based approaches in most tests. [48]

In a later paper [49], the authors of Logoot proposed an improved version of the algorithm. One of the changes they made was a new algorithm for the allocation of the integers in the elements' identifiers, which replaces `randomIntBetween` in line 14 in Figure 4.10. This new strategy chooses the integers randomly within a boundary and thereby limits the distance between two consecutive integers and therefore the corresponding identifiers. In doing so the *Boundary Strategy* groups the integers at the beginning of the available interval, near the preceding identifier, and leaves space for future insertions at the end which leads to shorter positions as this remaining space can be used for inserting new elements without extending the length of their positions. [49, 28]

4.2.5 LSEQ

Logoot belongs to the class of CRDTs that use variable-size identifiers[1]. Logoot's positions can grow indefinitely, as a result of which the algorithm's worst case space complexity is linear relative to the number of insert operations. This worst case can appear in the improved version of Logoot proposed in [49] if a user inserts characters at the beginning of the document and the calculated identifiers in the positions are always assigned the smallest available integers. Speaking in terms of the tree representation of the document, this means that for every new element a new level in the tree has to be created and therefore the space complexity is linear to the number of insert operations [28, 4]. A representation of such a tree can be seen in Figure 4.11 which shows the worst case Logoot structure after adding the characters `c`, `b` and `a` each at the start of the document resulting in the text `abc` and an unbalanced tree. If the performance deteriorates too much, the CRDT might have to employ a garbage collection algorithm that re-balances the positions, which can be directly translated to re-balancing the tree. Such algorithms use consensus protocols, requiring an unanimous decision in the network, with these protocols being costly. As this worst case scenario related to the insertion of characters at the beginning of the document suggests, Logoot's *Boundary Strategy* works best when users generally insert text at the end of the document as explained below. [28]

Making an assumption about the user's editing behavior is common for a CRDT. It makes allocating space for elements more efficient but leads to performance issues if the user does not behave as predicted. The authors of [28] therefore demand that an allocation

[1] A position in the Logoot algorithm can be classified as an identifier according to LSEQ's and h-LSEQ's definition of the term. As the application of LSEQ and h-LSEQ to Logoot will be evaluated in the following, Logoot's terms will be used apart from the term 'variable-size identifier' that describes a class of algorithms.

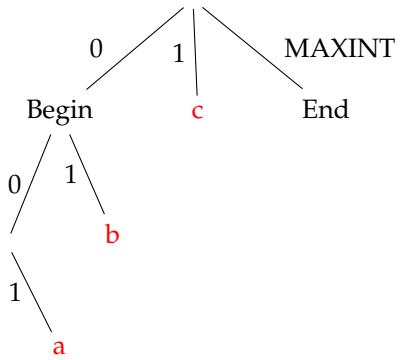

Figure 4.11: Tree representation of the worst case growth of positions in a Logoot document `abc` (based on [28, 26])

function embedded in a CRDT should be independent of the way the user edits the document. They see a fundamental problem in a CRDT trying to predict the user's actions as it only has knowledge about past and current operations and deducing future actions from this information is very complex. Another requirement for the allocation function in [28] is that it should have a sub-linear upper bound regarding its space complexity. This would limit the size of the positions to an acceptable length and make costly re-balancing protocols unnecessary. The polylogarithmic sequence (LSEQ) allocation function aims to meet the enumerated requirements [26]. LSEQ has two components, base doubling and strategy choice. [28]

Regarding the tree representation of the document in Figure 4.9, the base of a level in the tree refers to the arity of the nodes on that level. This means that a node on the level can have a maximum of *base* children. Regarding Logoot's identifiers, the base defines the range $[0, base]$ an identifier's integer is chosen from using the allocation strategy if the existence of other identifiers does not limit that range in any way. The goal is to adjust the base to the number of insert operations in that part of the document for an effective allocation of identifiers. [28]

A high base value results in a bigger range to choose integers for Logoot's identifiers from, which can be cost-effective if many insertions are performed in that part of the document as the spatial overhead of adding a new identifier to the positions, respectively adding a new level to the tree, is bigger than the overhead caused by longer integers in the identifiers. Vice versa, a small base value can also be profitable if only few insertions are performed. A constant base value, as the original and refined versions of Logoot use it, prevents the allocation function from taking full advantage of one of the boundary strategies described below as the available range to choose the integers from is limited. This results in longer spatially expensive positions and new levels in the tree representation of the document having to be created respectively. The goal resulting from these observations is to adjust the base according to the number of insertions. As the CRDT cannot calculate this number beforehand a good strategy is to begin with a small value for the base when the sequence is empty and then double it when a significant number of insertions occurs at a position, that is when a new level is created in the document tree. This leads to an exponential growth of the number of available identifiers on a certain level and makes up LSEQ's base doubling component. [28]

LSEQ stores the base for each depth and reuses it when a new level on that depth is created. Regarding the tree representation, the arity of a node is determined by its depth and is always twice the arity of the parent node, with the root node's arity being *base*. In

the original Logoot model the base is *MAXINT* on all levels. The base doubling component of LSEQ operates under the assumption that the creation of a new level in the tree representation of the document was caused by all integers on the previous level being in use. If the allocation strategy does not saturate the space before triggering the creation of a new level, the size of the positions grows rapidly as LSEQ doubles the base with each new depth leading to the identifiers on the newly created levels taking up more and more space. LSEQ's strategy choice component mitigates the effects of this worst case. [28]

The strategy choice component of LSEQ consists in randomly choosing a strategy among `boundary+` and `boundary-` for each level and saving that choice. If a new position between two other positions p and q is created using `boundary+`, which is Logoot's refined allocation strategy, the allocation function positions it between the preceding position p and $p + boundary$. Vice versa, if the `boundary-` strategy is used, the new position is placed between $q - boundary$ and the succeeding position q. Therefore, `boundary+` leaves space at the end of the position space and hence is better suited for users inserting text at the end of the document as the remaining space can be used to create new positions for these insertions without having to create a new level in the tree representation of the document respectively creating longer positions with more identifiers. Inversely, `boundary-` is better suited for users inserting text at the beginning of the document. Both keep the space complexity low if the user behaves as predicted by the strategy. The algorithms have opposing weaknesses and neither of them is an adequate choice for an allocation function on their own. [28]

The reasoning behind LSEQ's two components is that, as it is too complex to predict the user's editing behavior, the wrong strategy will be chosen for some levels of the document tree, but in the end the rewards will compensate the losses. Eventually, LSEQ will choose the right strategy, and as the base is doubled at each new level the gains of using the right strategy on some of the levels will more than outweigh the previous losses. [28]

The authors of LSEQ conducted a series of experiments on a single machine to evaluate the contribution of each part of the algorithm to its behavior in extreme cases and average setups. They concluded that the base doubling or random strategy choice components were not able to accomplish the desired sub-linear space complexity on their own, neither in the extreme nor the average case, but the combination of LSEQ's components was able to achieve the goal in both setups. The experiments were conducted using Logoot as a base with LSEQ as allocation function, as Logoot has the best performance of variable-size identifier CRDTs for sequences according to [4]. [28]

In conclusion, it can be said that LSEQ is a suitable allocation function for collaborative editing as it is adaptive, can handle various editing behaviors, and accomplishes sub-linear space complexity. It does not require a garbage collection protocol or any other way to re-balance the document tree. LSEQ achieves this performance using base doubling on each level of the tree in conjunction with random strategy choice between $boundary+$ and $boundary-$. [28]

4.2.6 h-LSEQ

LSEQ lowers the space-complexity of variable-size identifiers to a sub-linear level in local experiments, but it fails to achieve this goal in networks of multiple users with delivery of operations with latency. In those setups, LSEQ cannot guarantee an efficient allocation of the positions as without any agreement between the clients it is possible that they use opposing allocation strategies on a level of the document tree, which can lead to a quadratic growth of the positions. The authors of [27] observed the desired complexity in experiments using LSEQ when only a single user edited the document but the worst case when the document was edited by a group of 10 users. An agreement between all clients about which allocation strategy to use on each level is needed to restore LSEQ's original perfor-

mance by preventing them from using antagonist allocation strategies. As LSEQ's goal was to forgo protocols for establishing consensus in the network, a solution without additional exchange of information between the clients has to be found. The modification to LSEQ called h-LSEQ consists in replacing LSEQ's random strategy choice component by a strategy choice based on a hash function. The benefits of this approach are that it provides consensus between the clients without the overhead of an additional costly protocol or additional calculation and that it can take the place of the random strategy choice without changes to the rest of the LSEQ algorithm being necessary. [27]

On a single client choosing a strategy for the insertion of a new element into the CRDT's sequence takes three steps using h-LSEQ: The first step is to calculate the depth the new element will be added at. After that, the strategy choice function h is called with the depth as parameter. It returns the identifier of either the $boundary+$ or the $boundary-$ allocation strategy. This strategy identifier is then used to call the respective allocation function and retrieve a position for the new element. The strategy identifiers returned by the strategy choice function h follow a uniform distribution and therefore h does not favor any of the two allocation strategies. To achieve the desired consensus about the strategy choice between the clients, h-LSEQ requires an additional step compared to LSEQ before users can collaborate: The hash function h has to be initialized with a secret that all clients in an editing session share in order to get a unique hash function for the session. This results in each client's hash function returning the same allocation strategy identifier for a specified depth. [27]

On the network level, all users in one editing session use the same function h initialized with the same secret and the same mapping of the return value of h to an allocation strategy. This results in all instances of h-LSEQ choosing the same allocation strategy for the same depth. Therefore, a consensus has been established between the members of an editing session about which strategy to use at a specified depth. With the existence of this consensus the issue of clients utilizing opposing allocation strategies and hence not using the position space efficiently and increasing the space-complexity has been solved. [27]

The authors of [27] tested the efficiency of h-LSEQ in a series of experiments and confirmed that the addition of a hash function to the LSEQ framework suffices to achieve a sub-linear upper bound on the space complexity of a variable-size identifier sequence CRDT that was not reachable without the implicit a priori consensus about which allocation strategy to use at a depth. An evaluation of the impact of latency on the space complexity revealed that it had a positive influence on the length of positions and therefore is no concern in systems based on h-LSEQ. [27]

In a separate paper [26] the authors proved that LSEQ achieves a sublinear upper bound regarding the messages sent over the network with acceptable time and space complexity. They also developed a collaborative decentralized text editor that runs in the browser and used it to validate the described characteristics of h-LSEQ and their scalability using an experimental set-up with 600 connected browsers. [26]

In summary, h-LSEQ is an improved allocation strategy that scales LSEQ's sub-linear upper bound on space-complexity from a single user to a multiple user setting. It is not negatively affected by latency and does not require additional protocols compared to LSEQ. h-LSEQ can be used safely in collaborative editors to achieve competitive performance and scalability. [27]

4.2.7 Selected Algorithm

For the implementation of the collaborative text editor a CRDT was used as Operational Transformation algorithms have performance issues related to their use of vector clocks or tombstones and seem overly complex and Differential Synchronization also has performance issues and usability problems. Logoot was chosen as it is a fast CRDT according

to benchmarks in [4] and it was combined with h-LSEQ as allocation function as it limits Logoot's memory consumption.

CHAPTER 5

Implementation

This chapter gives an overview over the implementation of the collaborative text editor and challenges that arose during the implementation process. As the collaboration functionality is embedded into *Backstage 2*, the project's development stack was used, which is based on *JavaScript* as programming language with *Node.js*[1] as server-side runtime environment, *RethinkDB*[2] as database, and *React*[3] as front-end library.

5.1 General Procedure

The collaboration process starts with the lecturer logging into *Backstage 2*, creating teams which the students in the tutorial can join and then unlocking the collaborative editor for certain tasks that users can switch between. After the teams have been created, students can join a team as long as the maximum capacity of the team has not yet been reached. A user can only be a member of one team at a time, which they have to leave to join a new one and that they are taking part in during all tasks. Upon joining a team, the user can now participate in collaboration with their team members on tasks the lecturer has unlocked the collaborative editor for.

When the user Alice in Figure 5.1 edits the content of the editor, her actions are sent to a *collaboration*, which is an object that contains or is connected with all logic necessary for collaborative editing of one task. The *collaboration* passes the user's action on to the data structure used for that task. In most cases the data structure will send a message to the server via the *collaboration* and the `CollabStore` after processing the operation, as shown in Figure 5.1. The class `CollabStore` acts as a central sender and receiver of messages to or from the server. On the server the operation is received, forwarded to the clients of users that are members of the same team as the sender, and permanently saved in the database.

When the operation is received by the `CollabStore` on another user's client, as shown in Figure 5.2, it is passed on to the *collaboration* for the task the operation was executed on. Now the *collaboration* sends the operation to the data structure which will then initiate an update of the collaborative editor via the *collaboration*. This will lead to Alice's operation appearing in Bob's editor.

[1] `https://nodejs.org/`
[2] `https://www.rethinkdb.com/`
[3] `https://reactjs.org/`

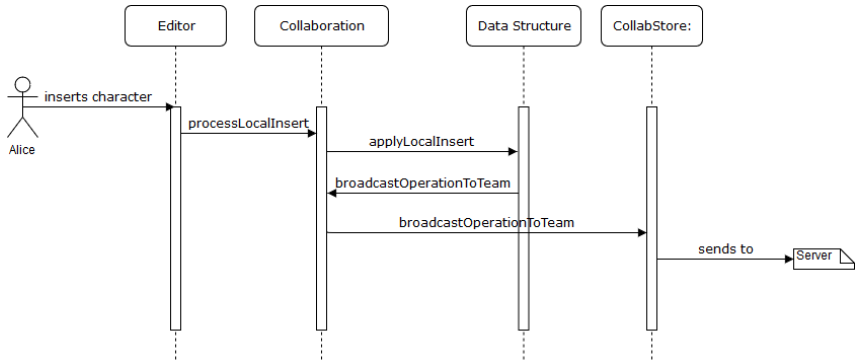

Figure 5.1: Simplified procedure for sending an edit to other users on a client

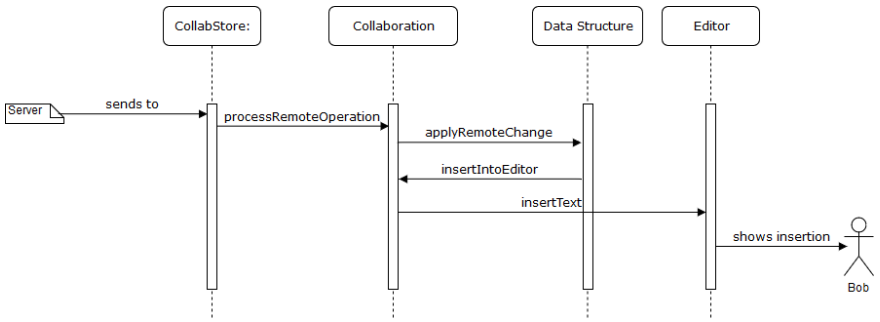

Figure 5.2: Simplified procedure for receiving an edit from another user on a client

5.2 Components Shared between Client and Server

Despite the collaborative editor being partly run on clients and the server, there is a portion of code that both client and server use of which the data structures are the most important part. The client needs access to the data structures to be able to save operations executed by the user and send data to the other users so that their clients can integrate the operations executed remotely. The server needs access to the data structures to save all operations in the database that it receives from a client and forwards to the sender's teammates' clients.

5.2.1 h-LSEQ Data Structure

As a new implementation of the chosen data structure, Logoot with h-LSEQ as an allocation function, would have been out of the scope of this thesis, an existing implementation[4] of Logoot combined with LSEQ was adapted. This implementation uses an adapted version of Logoot's algorithm for generating new positions between two existing ones (see Listing 4.10), which works analogous to the presented algorithm but only generates one position at a time. This algorithm was then combined with an h-LSEQ allocation function. Furthermore, the chosen implementation uses a two-dimensional array to store the positions instead of a linear array as proposed for the original Logoot algorithm. The purpose of this changed storage data structure is to resemble the structure a text document traditionally has in text editors divided into lines and columns, and therefore making the interaction of the editor and the data structure easier and less computationally expensive. An example of such an interaction is the insertion of a character by the user. The *Ace* editor

[4] `https://github.com/conclave-team/conclave`

used in the collaborative text editor fires a callback when the user inserts a character with the row and column the character is inserted at as parameters. Using the two-dimensional array as positions storage, the positions of the characters before and after the inserted one can be found much easier than using a linear array [12]. After these positions have been found, a position for the new character between them can be calculated and the character can be saved in the data structure.

5.2.2 SimpleSync Data Structure

The *SimpleSync* data structure which, like the CRDT data structure, is designed for text documents, serves as an example of how new synchronization algorithms can be added to the system. Instead of working with insert and delete operations like the CRDT based data structure, it stores and sends the complete document to the other users when a change is made. It works using a simple 'last edit wins' rule by sending the whole text of the document to the other users when the local user changes it, where it overwrites the text of the recipient.

5.3 Client

The client's main task is to show the collaborative editor to the user, to record interactions of the user with the editor and to process those by saving them in the local data structure and sending messages to the server containing those actions. Vice versa, the client is also responsible for receiving other users' actions, integrating them into the local data structure and updating the collaborative editor. Apart from the collaboration with others itself, the client also provides ways for the user to attend to other tasks like joining or leaving teams and submitting the solution of a task to the server.

5.3.1 CollabStore

The class `CollabStore` is a central part of the client. Its main responsibilities are:

- Management of teams (creation, deletion, adding/removing a user from a team).
- Sending messages to the server in order for it to execute actions like sending an operation to the other clients or sending synchronization data.
- Receiving messages from the server, potentially processing the result and then passing the received data to the class responsible for handling this kind of message. Most of the messages are passed to a *collaboration*.
- Managing a list of collaborations, one for each task.

To be able to manage the teams a user can join to collaborate with other users in that team, the `CollabStore` has a list with the data of the available teams and provides methods that other classes can call to execute actions related to these teams. This functionality is not included in the *collaborations* as it operates on a higher level in the sense that the teams are not specific to a single *collaboration*. The `CollabStore` receives updates for the team list from the server that it applies. Other classes and components can subscribe to updates of the team list and for example update the user interface when they happen.

The `CollabStore` also acts as the central access point for communication with the server for all components related to collaborative editing. While the task of sending messages to the server mostly consists in adding extra information to the payload, like the ID of the team the user is a member of, the receiving and distribution of messages to the classes

that are responsible for processing them is more complex. The server maintains a map of *collaborations* for this purpose.

5.3.2 Collaborations

A *collaboration* acts as an access point to all logic necessary for a successful real-time collaboration of the client on a task, either positioned in the collaboration itself or in other classes it delegates tasks to. An editor gets the collaboration for the task it is in after a request to the `CollabStore`. From then on the editor only interacts with the *collaboration* which executes or delegates all operations related to the client's side of the collaboration on the task. The responsibilities of the *collaboration* include:

- Synchronization management.
- The voting to submit the solution to a task to the server and therefore to the lecturer.
- Receiving messages related to the synchronization of a task from the `CollabStore`, processing those messages and calling methods of the editor to update its content based on the received data.
- The editor calls the collaboration's methods when the user executes an action in the editor and the collaboration then saves these operations in the data structure it has a reference to and generates a message that is sent to the other clients via the server so that they can 'replay' the operation.

A concrete *collaboration* extends the superclass `Collaboration` and implements the interface `ConcreteCollaboration`.

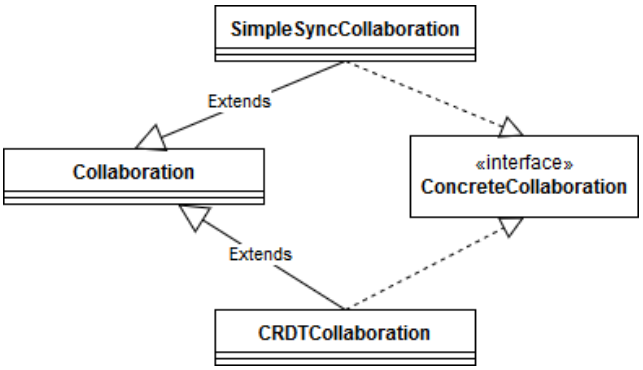

Figure 5.3: Class diagram for the *collaboration* classes/interface without methods or attributes

5.3.2.1 The Superclass `Collaboration`

The superclass `Collaboration`, as depicted in Figure 5.3, contains common features that are identical for all *collaborations*. These features include:

- Management of the *collaboration*'s synchronization status and therefore preventing synchronization issues.
- Management of the user's vote in submitting the *collaboration*'s content to the server. This includes storing and changing the user's vote status (for/against submitting the solution to the task to the server) and determining whether a majority for submitting a solution exists.

- Attributes all collaborations need like a reference to the editor the user uses. These include the information needed to authenticate with the `CollabStore`.
- Communication with the server. Messages for the other clients are passed to the `CollabStore`.

5.3.2.2 The Interface `ConcreteCollaboration`

All *collaborations*, like `CRDTCollaboration` in Figure 5.3, have to implement the interface `ConcreteCollaboration`. The interface `ConcreteCollaboration` contains, in addition to the methods a concrete *collaboration* should inherit from the class `Collaboration`, methods every collaboration is required to have but whose implementation can differ between the *collaborations*. The responsibilities of some of these methods include processing synchronization data from the server, processing operations from another client, sending operations to the server via the `CollabStore`, and resetting the *collaboration* to a specified state.

5.3.2.3 A Concrete Collaboration

A concrete collaboration implements or inherits all methods specified by the interface `ConcreteCollaboration`. Apart from that it also has unique methods for the combination of data structure and editor it manages. Examples of such methods are ones that the editor calls to pass actions of the user to the *collaboration*. In the case of the class `CRDTCollaboration` those methods are `processLocalInsert` and `processLocalDelete` as that *collaboration* is designed to transmit insert and delete operations. The class `SimpleSyncCollaboration` however, which also implements the interface `ConcreteCollaboration`, only has a method `processLocalContentChange` as it is designed to always operate on the whole text in the editor instead of insert and delete operations of characters.

5.3.3 Editor

As mentioned in the description of the *collaboration*, an editor is specifically designed for one concrete *collaboration* and therefore for one type of collaborative activity. In this implementation editors for collaboration on text were implemented but other applications like collaborative editing of tables, diagrams, or slides are also possible. Two editors were implemented for two different synchronization concepts.

5.3.3.1 CollabClime

`CollabClime` is the editor used with the CRDT-based synchronization strategy explained earlier. It extends the existing editor *Clime* that is used by *Backstage 2*. *Clime* is based on the *Ace*[5] editor which the added collaboration functionality mostly directly interacts with. The `CollabClime` editor captures insertions and deletions of characters by the user and then calls matching methods on its *collaboration* which uses the Logoot data structure with h-LSEQ as the allocation function. If a remote user inserts or deletes a character, the *collaboration* will eventually call a method of the local user's `ClimeEditor` to insert or delete the character locally.

[5] `https://ace.c9.io/`

5.3.3.2 CollabClimeSimple

`CollabClimeSimple` is the editor used with the *SimpleSync* data structure and is also based on the *Clime* editor and therefore also on the *Ace* editor. Unlike the `CollabClime` editor the `CollabClimeSimple` editor only generally captures when the user changes the text in the editor and sends the whole updated text to its *collaboration* via a method call. The *collaboration* that uses the described *SimpleSync* data structure then proceeds to send the updated content to the server. Apart from the different editor, collaboration and data structure, all other components of the system are used by this editor the same way as by the `CollabClime` editor which shows the adaptability of the system.

5.4 Server

The main task of the server component is to forward messages it receives from a user to that user's teammates. Apart from that, it also saves every operation it forwards in the database which allows it to send the current state of a data structure to a user, for example when they reload the web page. Furthermore, this permanent storage of operations allows the system to retain the state of data structures even if all users leave a team, and the data structure therefore is no longer present on any client, or the server is rebooted and the data structure is erased from the server's RAM.

5.4.1 Collaboration Socket

The object `collaborationSocket` is the main server component of the collaborative editor. It is responsible for receiving messages from the client, sending messages back to the sending client or to other clients, and processing the received messages. The purposes of the messages include the creation of teams, joining or leaving a team, requesting synchronization with the server, voting to submit a solution to a task, and sending an operation executed in a collaborative editor to other clients.

5.4.2 Worker

The worker is responsible for storing every operation a client executes safely in the database on the server. This functionality was outsourced from the main thread to prevent blocking the receiving and sending of messages from/to the clients with the long procedure of restoring a data structure's state from the database, replaying an operation using that data structure, and saving the state back to the database. The worker is implemented using a *Worker Thread* using *Node.js' worker* module, which "provides a way to create multiple environments running on independent threads, and to create message channels between them" [29].

5.4.3 Worker Queue

The class `WorkerQueue` is responsible for ensuring that the worker only executes one operation at a time in the order the messages containing the operations arrived on the server.

If a message arrives and the queue contains more than one operation, the new operation is added to the end of the queue. If there is no operation in the queue, the new operation is added to the queue and then sent to the worker. The intuitive way of handling the queue would be to directly send the operation to the worker if the queue is empty without adding it. This approach results in the possibility of more operations being sent to the worker while it is already processing an operation, because there is no indicator for the worker being busy that prevents more operations to be sent to it. After the worker receives

an operation, it proceeds as described above: It gets the data structure's state from the database, initializes a data structure with it, executes the received operation on the data structure and saves the updated state of the data structure back to the database. After that, the worker sends a message to the main thread, which then tells the `WorkerQueue` to send the next operation to the worker. The `WorkerQueue` removes the first operation from the queue and sends the next operation in the queue to the worker.

The class `WorkerQueue` was introduced because the initial implementation of the worker had significant consistency issues. Operations were sent to the worker when the main thread received them. The worker then proceeded to get the state for the data structure the operation should be applied on from the database. Problems arose when a large number of operations were sent to the worker consecutively. It received the operations from the main thread in the correct order but then started to process them in a non-deterministic order. Together with some of the database queries returning stale states of the database this led to problems like characters being inserted at the same position a previous character was inserted at and therefore operations being lost. The result was that an invalid data structure was saved in the database, which was sent to the clients for example when they reloaded the page.

5.4.4 Synchronization Request Queue

The class `SyncRequestQueue` maintains a queue of synchronization requests. The queue is saved in the form of a map which stores for every combination of team ID and task ID a queue of synchronization requests for the task and a counter that represents the number of operations on the task that are currently in the `WorkerQueue`.

When a synchronization request arrives, the main thread sends it to the `SyncRequestQueue` which checks whether an entry for the combination of team ID and task ID exists in the map. If there is no such entry, the synchronization request is answered immediately but if an entry exists, meaning that there are currently operations for the requested task in the `WorkerQueue`, the synchronization request is added to that entry's queue. When an operation arrives at the server from a client, the `WorkerQueue` notifies the `SyncRequestQueue` and the counter of the matching entry in the map is increased or a new entry is created. When an operation is removed from the `WorkerQueue`, the `SyncRequestQueue` is also notified and the counter of the entry of the task the operation was executed on is either decreased or if it is `1`, which indicates that the executed operation was the last one in the queue for that task, all synchronization requests in the queue of the entry are executed and the entry is deleted from the map.

The class `SyncRequestQueue` was introduced as a fix for an issue that arose when a user sent a large number of operations to the server, for example by inserting a long text into the editor from their computer's clipboard. The large number of requests the worker thread had to process kept it busy for a significant amount of time. If another user's client sent a synchronization request to the server for the same task, the worker was still busy processing operations for, for example because that user just joined the team, it could happen that the request was answered with outdated synchronization data. This happened if the main thread processing the synchronization request accessed the database but the worker was not done storing operations in the database yet for the requested task and therefore the main thread retrieved an outdated state from the database that would later be overwritten.

5.4.5 Optimization of the Worker

Another issue that arose in connection with the worker was that the worker was slow in processing and saving the operations in the database compared to the speed the main

thread forwarded the operations to the clients with. In combination with the *SyncRequestQueue* this could lead to users having to wait a long time before their clients could synchronize with the server. To solve this problem the worker keeps references to all data structures it has already executed operations on in a map. After executing operations on one of them it saves the data structure's new state in the database to permanently store it. This significantly decreased the waiting time for the processing of synchronization requests as in many cases the worker can reuse a data structure from the map and therefore from RAM instead of retrieving it from the database which is slower.

5.5 Extensibility of the System

As the addition of the *SimpleSync* data structure with the matching editor shows, adding a new data structure or a new type of collaboration is easy. To add a new type of content to collaborate on, like for example tables instead of text, a data structure would have to be provided and it would have to support a few basic features needed for it to work with the worker on the server. Then a new *collaboration* would have to be added that extends the `Collaboration` class, implements the `ConcreteCollaboration` interface and can interact with the new editor by receiving operations from it and applying remote operations to its content. Furthermore, a new editor would have to be added that shows the new type of content and can communicate with the new *collaboration* it will get a reference to. Apart from that, the existing structures for communication, storage in the database, team management, voting to submit a solution and synchronization could be used. As adding a new type of content to collaborate on is the most sophisticated change that can be made to the system, replacing or adding other parts of it is also easy.

CHAPTER 6

Conclusion and Outlook

In this thesis a concept to complement oral communication during group work in university courses with a collaborative text editor was introduced after examination of reports about the usage of such editors in teaching. Results of the examination of these usages included perception of better teamwork among the students [20, 52], better quality of results [43], and a positive attitude towards the editor [18] and the taught subject [11]. An exemplary application of the concept to the learning management system *Backstage 2* was described and the psychological foundations of the concept examined. Furthermore, the challenges connected to the simultaneous editing of a common document were demonstrated and two general solutions to these challenges, optimistic replication and pessimistic replication, were compared. Optimistic replication was chosen because it allows for better availability of the shared document and the weaker consistency guarantees it offers are sufficient for the application in group work at a university. After that, a number of algorithms from the optimistic replication category were compared leading to the choice of Logoot combined with h-LSEQ as an allocation function as the algorithm for an exemplary implementation of the proposed concept to enable collaboration on texts. Lastly, the way the proposed concept was implemented and embedded in *Backstage 2* was described.

The implementation, as provided as part of this thesis, does not contain all elements of the proposed concept, as the component that allows the lecturer to view and present the solutions submitted by the students has not been implemented. An extension of the existing implementation with such a component would complete the system and allow for its usage in practice. Furthermore, modifications to the system to make it suitable for work on homework assignments without the students sitting next to each other in the classroom could be made. These modifications could include adding some mean of communication for students working on the same task from home, like a live chat, and changing the way users join teams to only allow invited users to join a team. This would prevent students from only joining another team to copy its solution to a task and then leave the team again. Another possible change to the system would be making the teams permanent. This would make deeper connections between the members of a team possible and could lead to them helping each other with learning and the tasks they are presented with. Based on these, permanent team gamification elements, like a competition between teams throughout the semester where they are awarded points for excellent solutions to tasks, could be integrated. In [23] a similar gamification element focusing on quizzes was built to improve the students' participation in lectures and an evaluation revealed that it accomplished that

goal. Apart from that, anonymised collaboration protocols could be created with the consent of the users. These protocols could be made available to developers of synchronization algorithms to be used in benchmarks. As they represent real time communication, this could lead to the tests being closer to the usage of the algorithms in practice than tests using the edit history of *Wikipedia* articles [48, 28] or synthetic operations [27] that are used in some benchmarks. Additionally, a user study could be conducted to examine the reception of the current or an extended version of the system by the students. It could be used to verify that the usage of the collaborative editor during group work has a positive impact on the students' work and to gather information about how to further improve the system.

Bibliography

[1] *Google Docs, Sheets, and Slides*, https://en.wikipedia.org/wiki/Google_Docs,_Sheets,_and_Slides, downloaded on 03.09.2018.

[2] Lakmal Abeysekera and Phillip Dawson, *Motivation and cognitive load in the flipped classroom: definition, rationale and a call for research*, Higher Education Research & Development **34** (2014), no. 1, 1–14.

[3] Paul Adams, *Exploring social constructivism: theories and practicalities*, Education 3-13 **34** (2006), no. 3, 243–257.

[4] Mehdi Ahmed-Nacer, Claudia-Lavinia Ignat, Gérald Oster, Hyun-Gul Roh, and Pascal Urso, *Evaluating CRDTs for Real-time Document Editing*, 11th ACM Symposium on Document Engineering (Mountain View, California, United States) (ACM, ed.), September 2011, pp. 103–112.

[5] Maryam Alavi, *Computer-Mediated Collaborative Learning: An Empirical Evaluation*, MIS Quarterly **18** (1994), no. 2, 159–174.

[6] Stuart J. Beatty, Katherine A. Kelley, Anne H. Metzger, Katherine L. Bellebaum, and James W. McAuley, *Team-based Learning in Therapeutics Workshop Sessions*, American Journal of Pharmaceutical Education **73** (2009), no. 6, 100.

[7] Jacob Lowell Bishop and Matthew A. Verleger, *The flipped classroom: A survey of the research*, ASEE National Conference Proceedings, Atlanta, GA, vol. 30, 2013, pp. 1–18.

[8] Ina Blau and Avner Caspi, *What type of collaboration helps? Psychological ownership, perceived learning and outcome quality of collaboration using Google Docs*, Proceedings of the Chais conference on instructional technologies research, vol. 12, 2009, pp. 48–55.

[9] Cornelia Brodahl, Said Hadjerrouit, and Nils Kristian Hansen, *Collaborative writing with Web 2.0 technologies: education students' perceptions*, Journal of Information Technology Education: Innovations in Practice **10** (2011), 73–103.

[10] Cornelia Brodahl and Nils Kristian Hansen, *Education students' use of collaborative writing tools in collectively reflective essay papers*, Journal of Information Technology Education: Research **13** (2014), 91–120.

[11] Lurdes Cardoso and Clara Pereira Coutinho, *Web 2.0 learning environments in vocational education: a study on the use of collaborative online tools in the Statistics module*, Proceedings of Society for Information Technology & Teacher Education International Conference 2011 (Nashville, Tennessee, USA) (Matthew Koehler and Punya Mishra, eds.),

Association for the Advancement of Computing in Education (AACE), March 2011, pp. 3155–3164.

[12] conclave team, *Conclave - A private and secure real-time collaborative text editor*, `https://conclave-team.github.io/conclave-site/`, downloaded on 28.04.2018.

[13] Clarence A. Ellis and Simon J. Gibbs, *Concurrency control in groupware systems*, Acm Sigmod Record, vol. 18, ACM, 1989, pp. 399–407.

[14] Neil Fraser, *Differential synchronization*, Proceedings of the 9th ACM symposium on Document engineering, ACM, 2009, pp. 13–20.

[15] Anuradha A. Gokhale, *Collaborative Learning Enhances Critical Thinking*, Journal of Technology Education **7** (1995), no. 1.

[16] Google, *Google Docs - create and edit documents online, for free.*, `https://www.google.com/intl/en_us/docs/about/`, downloaded on 14.05.2018.

[17] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli, *Formal design and verification of operational transformation algorithms for copies convergence*, Theoretical Computer Science **351** (2006), no. 2, 167–183.

[18] Kyeong-Ouk Jeong, *A Study on the Integration of Google Docs as a Web-based Collaborative Learning Platform in EFL Writing Instruction*, Indian Journal of Science and Technology **9** (2016), 39.

[19] Samuel Kai-Wai Chu and David M. Kennedy, *Using online collaborative tools for groups to co-construct knowledge*, Online Information Review **35** (2011), no. 4, 581–597.

[20] Greg Kessler, Dawn Bikowski, and Jordan Boggs, *Collaborative writing among second language learners in academic web-based projects*, Language Learning & Technology **16** (2012), no. 1, 91–109.

[21] Ravern Koh, *Collaborative text editing with Logoot*, `https://medium.com/@ravernkoh/collaborative-text-editing-with-logoot-a632735f731f`, downloaded on 17.06.2018.

[22] Paul G. Koles, Adrienne Stolfi, Nicole J. Borges, Stuart Nelson, and Dean X. Parmelee, *The Impact of Team-Based Learning on Medical Students' Academic Performance*, Academic Medicine **85** (2010), no. 11, 1739–1745.

[23] Sebastian Mader and François Bry, *Gaming the Lecture Hall: Using Social Gamification to Enhance Student Motivation and Participation*.

[24] Larry K. Michaelsen and Michael Sweet, *The essential elements of team-based learning*, New Directions for Teaching and Learning **2008** (2008), no. 116, 7–27.

[25] ———, *Team-based learning*, New Directions for Teaching and Learning **2011** (2011), no. 128, 41–51.

[26] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui, *A scalable sequence encoding for collaborative editing*, Concurrency and Computation: Practice and Experience (2017).

[27] Brice Nédelec, Pascal Molli, Achour Mostéfaoui, and Emmanuel Desmontils, *Concurrency Effects Over Variable-size Identifiers in Distributed Collaborative Editing*, Document Changes: Modeling, Detection, Storage and Visualization (Florence, Italy), CEUR Workshop Proceedings, vol. 1008, September 2013, pp. 0–7.

[28] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils, *LSEQ: An Adaptive Structure for Sequences in Distributed Collaborative Editing*, 13th ACM Symposium on Document Engineering (DocEng) (New York, NY, USA), DocEng '13, ACM, September 2013, pp. 37–46.

[29] Node.js Foundation, *Node.js v10.8.0 documentation*, `https://nodejs.org/dist/latest-v10.x/docs/api/worker_threads.html`, downloaded on 13.08.2018.

[30] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine, *Proving correctness of transformation functions in collaborative editing systems*, Research Report RR-5795, INRIA, 2005.

[31] ———, *Data Consistency for P2P Collaborative Editing*, ACM Conference on Computer-Supported Cooperative Work - CSCW 2006 (Banff, Alberta, Canada), Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, ACM Press, 2006, pp. 259–268.

[32] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leția, *A commutative replicated data type for cooperative editing*, 29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009) (Montreal, Québec, Canada), IEEE Computer Society, June 2009, pp. 395–403.

[33] Michel Raynal and Mukesh Singhal, *Logical Time: A Way to Capture Causality in Distributed Systems*, Research Report RR-2472, INRIA, 1995.

[34] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser, *An Integrating, Transformation-oriented Approach to Concurrency Control and Undo in Group Editors*, Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work (New York, NY, USA), CSCW '96, ACM, 1996, pp. 288–297.

[35] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee, *Replicated abstract data types: Building blocks for collaborative applications*, Journal of Parallel and Distributed Computing **71** (2011), no. 3, 354 – 368.

[36] Kristie W. Ruddick, *Improving chemical education from high school to college using a more hands-on approach*, Ph.D. thesis, 2012, p. 111.

[37] Yasushi Saito and Marc Shapiro, *Optimistic Replication*, Technical report, Microsoft Research, 2003.

[38] Nachiket Shankar and R. Roopa, *Evaluation of a modified team based learning method for teaching general embryology to 1styear medical graduate students*, Indian Journal of Medical Sciences **63** (2009), no. 1, 4–12.

[39] Marc Shapiro and Nuno Preguiça, *Designing a commutative replicated data type*, Research Report RR-6320, INRIA, 2007.

[40] Jeremy F. Strayer, *How learning in an inverted classroom influences cooperation, innovation and task orientation*, Learning Environments Research **15** (2012), no. 2, 171–193.

[41] Chengzheng Sun and Clarence Ellis, *Operational Transformation in Real-time Group Editors: Issues, Algorithms, and Achievements*, Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (New York, NY, USA), CSCW '98, ACM, 1998, pp. 59–68.

[42] David Sun, Steven Xia, Chengzheng Sun, and David Chen, *Operational Transformation for Collaborative Word Processing*, Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (New York, NY, USA), CSCW '04, ACM, 2004, pp. 437–446.

[43] Ornprapat Suwantarathip and Saovapa Wichadee, *The Effects of Collaborative Writing Activity Using Google Docs on Students' Writing Abilities*, Turkish Online Journal of Educational Technology-TOJET **13** (2014), no. 2, 148–156.

[44] The Etherpad Foundation, *Etherpad*, `http://etherpad.org`, downloaded on 14.05.2018.

[45] Roxanne Toto and Hien Nguyen, *Flipping the Work Design in an industrial engineering course*, 2009 39th IEEE Frontiers in Education Conference, IEEE, oct 2009.

[46] Mark Warschauer, *Computer-Mediated Collaborative Learning: Theory and Practice*, The Modern Language Journal **81** (1997), no. 4, 470–481.

[47] Warren E. Watson, Larry K. Michaelsen, and Walt Sharp, *Member competence, group interaction, and group decision making: A longitudinal study*, Journal of Applied Psychology **76** (1991), no. 6, 803–809.

[48] Stéphane Weiss, Pascal Urso, and Pascal Molli, *Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks*, 29th IEEE International Conference on Distributed Computing Systems - ICDCS 2009 (Montreal, Canada), 2009 29th IEEE International Conference on Distributed Computing Systems, IEEE, June 2009, pp. 404–412.

[49] ———, *Logoot-Undo: Distributed Collaborative Editing System on P2P Networks*, IEEE Transactions on Parallel and Distributed Systems **21** (2010), no. 8, 1162–1174.

[50] Wayne Roy Wilson, *The use of permanent learning groups in teaching introductory accounting*, Ph.D. thesis, Michael F. Price College of Business, 1982.

[51] Sarah Zappe, Robert Leicht, John Messner, Thomas Litzginer, and Hyeon Woo Lee, *"Flipping" the classroom to explore active learning in a large undergraduate course*, ASEE Annual Conference and Exposition, Conference Proceedings, June 2009.

[52] Wenyi Zhou, Elizabeth Simpson, and Denise Pinette Domizi, *Google Docs in an Out-of-Class Collaborative Writing Activity*, International Journal of Teaching and Learning in Higher Education **24** (2012), no. 3, 359–375.